Math. Struct. in Comp. Science (2004), *vol.* 14, *pp.* 421–468. © 2004 Cambridge University Press
DOI: 10.1017/S0960129504004189 Printed in the United Kingdom

Concurrent models for Linda with transactions[†]

ROBERTO BRUNI and UGO MONTANARI

Dipartimento di Informatica, Università di Pisa, Via F. Buonarrot, m 2, 56127 Pisa, Italy
{bruni,ugo}@di.unipi.it

Received 11 March 2002; revised 30 March 2003

Nowadays concepts, languages and models for coordination cannot leave aside the needs of the increasing number of commercial applications based on global computing over the (inter)net. For example, platforms like Microsoft .NET and Sun Microsystems Java come equipped with packages for supporting *ad hoc* transactional features, which are essential for most business applications. We show how to extend the coordination language *par excellence* (*viz*. Linda) with basic primitives for transactions, while retaining a formal model for its concurrent computations. This is achieved by exploiting a variation of Petri nets, called zero-safe nets, where transactions can be suitably modelled by distinguishing between stable places (ordinary ones) and zero places (where tokens can only be temporarily allocated, defining hidden states). The relevance of the transaction mechanism is illustrated in terms of expressive power. Finally, it is shown that stable places and transactions viewed as atomic steps define an abstract semantics that is apt for a fully algebraic treatment, as demonstrated *via* categorical adjunctions between suitable categories of nets.

1. Introduction

Linda (Gelernter 1985) can be considered as the most representative language for coordination, where (asynchronous) communication is obtained by inserting and withdrawing elements to and from a shared multiset of tuples (that is, messages, data), called *tuple space*, where structured data are also allowed. More precisely, Linda agents are built on the following primitives:

(1) $out(a)$ for the output of the message $\langle a \rangle$ on the tuple space;
(2) $rd(a)$ for the reading of a message from the tuple space (but without consuming it);
(3) $in(a)$ for the fetching of a message, after which the message is no longer available in the tuple space.

Two additional predicates $rdp(a)$ and $inp(a)$ allow, respectively, for:

(4) checking for the presence of a message without consuming it;
(5) atomically testing for, and then consuming the message if present.

[†] Research supported by the FET-GC Project IST-2001-32747 AGILE and by the Italian MIUR Project COFIN 2001013518 COMETA. The first author is also supported by an Italian CNR fellowship for research on Information Sciences and Technologies, and by the Computer Science Department of the University of Illinois at Urbana-Champaign.

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

The two predicates can be used as the boolean components of conditional constructs. The communication mechanism of Linda is called *generative* because, once generated and until removed, each message becomes equally accessible to all processes without being bound to a particular one.

These operations make Linda suitable for expressing non-atomic process coordination in a concurrent setting, and they have been exploited, together with suitable extensions, as primitives for distributed language implementations, such as KLAIM (De Nicola *et al.* 1998), the coordination middlewares JavaSpaces (Sun Microsystems 2000) from Sun and Tspaces (Wyckoff *et al.* 1998) from IBM. However, there is no primitive for expressing transactions in the core language, and it seems that, for languages based on generative communications, the problem of finding a satisfactory way to program transactions is currently receiving a lot of attention, see, for example, the extension in Anderson and Shasha (1991) or the one proposed in Busi and Zavattaro (2001) for studying *serialisability* in JavaSpaces, which were both largely inspired by the traditional way of programming transactions in databases *via* create/commit primitives for starting and closing transactions and based on the (centralised) two-phase commit protocol (2PC).

In this paper we propose an alternative approach, inspired by a variation of Petri nets that comes equipped with a special kind of data-driven synchronisation between transitions. The advantages of this approach with respect to more traditional ones are that:

(1) multiway transactions can be naturally programmed (that is, transactions with many entry and exit points);
(2) the coordination layer, and especially the commit protocol, can be completely distributed among the participants;
(3) both the refined view (where the actions composing the transactions are visible) and the abstract view (where transactions are seen as atomic operations) are formally defined and related.

The key observation is that when tuples are seen as non-structured data, the concurrent semantics of Linda can be suitably modelled *via* Petri nets with read and inhibitor arcs (Busi *et al.* 1996; Busi 1998). The net modelling exploits two basic features:

(i) states are modelled as multisets of tuples and agents;
(ii) elementary actions (transition firings) can atomically fetch (and release) the agents and the object tuples.

A main advantage of these assumptions is that the notion of concurrent firing (called a *step*) comes as a consequence of the multiset structure of states, that is, computations can be straightforwardly equipped with a truly concurrent semantics (as opposed to interleaving semantics).

However, ordinary place/transition Petri nets (PT nets) are not expressive enough to faithfully model all the operations of Linda and two further enrichments are needed:

— *nets with read arcs* (r-nets) allow for modelling 'read without consume', where many readers can access concurrently the same resource (whereas in ordinary PT nets these operations are usually rendered *via* self-loops);

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

— *nets with inhibitor arcs* (i-nets) allow for testing the 'absence of tokens', an operation that makes the framework Turing complete, even under the assumption of finiteness of the net (as opposed to PT nets and r-nets, where reachability and deadlock are decidable).

Read and inhibitor arcs can be easily combined together to model a general concept of positive/negative *context*. Note that in the literature, the terms *contextual nets* and *c-nets* are abused in denoting either just positive contexts or both kinds of context; to avoid confusion, we stick to the unambiguous notation *ri-nets* for nets where both read and inhibitor arcs are allowed.

Regarding the use of inhibitor arcs in the net models of Linda processes, the distinction between the *ordered* and *unordered* semantics of Linda-like process algebras drawn in Busi *et al.* (2000a), Busi *et al.* (2000b) and Busi (1998) is interesting: in the former semantics, the output of a tuple is seen as the atomic execution of the emission of the message followed by its rendering in the tuple space (making it available to other agents); whereas in the latter view these two phases are made autonomous (emitted messages can be rendered asynchronously at any later time, disregarding the order in which they have been emitted, hence the name 'unordered'). This distinction has important consequences in terms of language expressiveness: the ordered semantics originates a Turing complete language, whilst, in the unordered case, deadlock is decidable (Busi *et al.* 2000b; Busi 1998). As a consequence, inhibitor arcs are necessary for modelling the ordered semantics, while in the unordered case it is indeed possible to define a deadlock-equivalent net semantics based on finite PT nets, as demonstrated in Busi *et al.* (2000b) and Busi (1998).

While PT nets rely on place synchronisation, they lack synchronisation of transitions, which would allow us to regard some (finite, but possibly complex and concurrent) computations as atomic events. Transition synchronisation is, for example, essential in modelling distributed decision algorithms, where no particular location can implement a choice point by itself. Our observation is probably not surprising, since the necessity of defining entities in isolation that can cooperate by sending and receiving information, but are otherwise seen like black boxes from the environment, was already evident in Dijkstra communicating processes and in process algebras. Transition synchronisation is also a key aspect of open-ended systems: When designing the processes in isolation, the programmer must consider all the interactions that can take place, but these may well depend on the received data and may require some atomicity assumptions on their execution (think, for example, of communication protocols, with transmission requests and acks, or of e-commerce applications, where customer's payment and goods/service delivery must be either both guaranteed or both cancelled when the transaction ends). We denote atomic computations in a concurrent scenario by the term *transactions* (abusing databases terminology). Thanks to the analogy with PT nets (tuples as tokens), the idea is to provide a way for programming transactions in Linda by exploiting the transactional extension of PT nets proposed in Bruni and Montanari (2000b):

— *zero-safe nets* (ZS nets) introduces a simple mechanism for synchronising transitions by producing and retrieving tokens from suitable places;

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

ZS nets exploit this idea by distinguishing between *stable* places (the ordinary repositories for resources, defining observable states) and *zero* places that cannot contain tokens in any observable state. The firing of a transition will possibly put tokens in zero places, beginning a transaction; these tokens (called *zero tokens*) are used to coordinate the transaction. All zero tokens must be removed to commit the transaction. Moreover, all the stable tokens produced during the transaction are effectively released only when the transaction ends. Thus, all the stable resources fetched during the transaction must be present in the initial stable marking. This describes, to some extent, the low-level model. At the abstract level, transactions become ordinary transitions. This viewpoint yields a PT net A_B, which is the abstract counterpart of the ZS net B: the places of A_B are the stable places of B, and each transition of A_B corresponds to an elementary transaction of B (that is, a transaction that cannot be decomposed into two smaller disjoint transactions).

Note that A_B can also become infinite when B is finite, and that transactions retain all the causal and concurrent information about the evolution of B. Moreover, distributed implementations for ZS nets have been proposed in Bruni and Montanari (2000a) and Bruni *et al.* (2002a), based, respectively, on a distributed interpreter that exploits the ordinary net unfolding and on a suitable encoding in the Join calculus that exploits a novel distributed commit protocol. Hence, ZS nets can be used, for example, to give a modular presentation of distributed decision making, as any PT net can be modelled as the abstract counterpart of a free choice ZS net (Bruni and Montanari 2000a). It is worth noting that the distributed commit protocol is compiled in the encoding and thus it is completely transparent to the programmer.

At the level of the coordination language, the distinction between stable places and zero places corresponds to distinguishing a class of *low-level* messages whose only role is to coordinate the exchange of services or information between agents. We call these messages *zero-safe*, because it only makes sense for them to exist inside a transaction, that is, their lifetime lasts as long as necessary for their producers and consumers to agree on some decision. The remaining messages are called *stable* to make clear that the information they contain contributes substantially to the actual configuration of the system. Moreover, we introduce a second kind of prefixing, called *atomic prefixing*, that allows for grouping sequences of actions to be performed inside a transaction. We show that this is consistent with the net modelling in Busi *et al.* (1996) and Busi (1998), where, in fact, processes are also represented by places, and as such can be separated into two classes (stable and zero-safe). We also show that without the atomic prefixing only two-way synchronisation is possible, and is between exactly one output and one input operation.

As a simple example, let us consider the modelling of multiset rewriting: an agent P needs all the messages $a_1, \ldots, a_n$, regardless of the order in which they are consumed, to start a task Q. Of course, we look for a simple way of specifying that no process P' can in the meantime fetch some of the resources $a_1, \ldots, a_n$, thus deadlocking P. The idea is to write P as the parallel composition of $n+1$ processes $in(a_1)_\, out(z).0$, $in(a_2)_\, out(z).0$, $\ldots$, $in(a_n)_\, out(z).0$, and $in(z)_\, \ldots_\, in(z)_\, Q$, where: all the a_i's are stable; z is zero-safe; and $_$ denotes the atomic prefixing, with Q prefixed n times by $in(z)$. Then, the transaction starts as soon as any a_i is consumed and the atomic prefixing forces the output of a zero-safe message z; moreover, the transaction can only commit when n messages z have

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

been produced, which the atomic prefix of Q can consume, otherwise the transaction will fail, restoring the initial state. Hence Q cannot be activated unless all the a_i's have been retrieved.

We call TraLinda the language obtained by extending the kernel of Linda with the above illustrated primitives for transactions (atomic prefixing and the typing mechanism for partitioning messages in two classes, that is, zero-safe and stable).

The net modelling of TraLinda thus includes read and inhibitor arcs for dealing with ordinary Linda primitives, and zero places to express transactions. While these features can seem, to some extent, orthogonal flavours, we show that this is not the case and that some restriction must be imposed to maintain the consistency between the abstract and low-level views.

A fully algebraic account of ZS r-nets can be obtained by exploiting universal constructions between theories in partial membership equational logic, thus characterising the abstract view as a coreflection. Though the construction can hardly be extended to ZS ri-nets, we argue that having a nice mathematical way of reconciling the refined and abstract view (*via* universal constructions) is still important, as ZS r-nets are sufficient for modelling the 'positive' primitives of TraLinda (that is, those whose semantics does not involve negative premises).

Related work

The use of inhibitor arcs and read arcs (introduced separately in Christensen and Hansen (1993), Montanari and Rossi (1995), Janicki and Koutny (1995) and Vogler (1997a), and also known as *condition-arcs*, or *test-arcs*) is crucial to our approach: read arcs allow for *reading* resources *without consuming* them, which is the natural way of reusing a stable token inside a transaction, granting multiple concurrent accesses to the same resource; inhibitor arcs increase the expressiveness of the framework providing *test for absence*. Contextual nets have found applications to, for example, transaction serialisability in databases (De Francesco *et al.* 1994), concurrent constraint programming (Montanari and Rossi 1994), and asynchronous systems (Vogler 1997b).

The formalisation of Linda-like coordination languages as process algebras was initiated in Busi *et al.* (1998; 2000a), and has recently been extended in Busi *et al.* (2000c) and Busi and Zavattaro (2001) to deal with primitives for *event notification*, *create-commit transactions*, *timeout*, and *leasing*, inspired by the proposals of JavaSpaces and TSpaces.

The main references for zero-safe nets are Bruni and Montanari (2000b), where their operational and abstract semantics under the collective and individual token philosophies are presented and compared, and Bruni and Montanari (2000a) and Bruni *et al.* (2002a), where distributed implementations are discussed, based on net unfolding rules and distributed Join calculus, respectively. Zero-safe nets with read arcs are considered in (Bruni and Montanari 2001a), extending the approach of (Bruni and Montanari 2000a).

This paper is the full version of Bruni and Montanari (2001b), where zero-safe nets with inhibitor arcs were first considered and where the language TraLinda and its net semantics where somewhat informally described. Most examples, most technical results

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

and the whole part on the algebraic presentation of the abstract semantics of zero-safe r-nets are entirely original to this contribution.

Our net semantics for TraLinda is based on the net models for Linda proposed in Busi *et al.* (1996) and Busi (1998). The comparison with the CHAM-like semantics of JavaSpaces proposed in Busi and Zavattaro (2001) is also original to this contribution.

Structure of the paper

In Section 2, we recall the basics of Petri nets with read arcs, inhibitor arcs and zero places. Section 3 recalls the syntax and the concurrent semantics of Linda as defined in Busi *et al.* (1996) and Busi (1998). Section 4 is the core of the paper and is divided into three parts: Section 4.1 discusses the introduction of inhibitor arcs in ZS r-nets; Section 4.2 defines the language TraLinda and its concurrent semantics together with several examples; Section 4.3 shows how to recast the difference between the ordered and unordered semantics of Linda in terms of the models for TraLinda; Section 4.4 informally discusses a distributed implementation for TraLinda. Section 5 explains the main differences with respect to the transaction mechanisms proposed in Bruni *et al.* (2002a) for BizTalk and in Busi and Zavattaro (2001) for JavaSpaces. Section 6 is devoted to the algebraic semantics of zero-safe r-nets.

2. Petri nets

A *net* is a triple $N = (S_N, T_N, \mathsf{F}_N)$, where S_N is the set of *places* $a, a', \ldots$, T_N is the set of *transitions* $t, t', \ldots$ (with $S_N \cap T_N = \emptyset$) and $\mathsf{F}_N \subseteq (S_N \times T_N) \cup (T_N \times S_N)$ is called the *flow relation*.

The elements of F_N are called *arcs*, and we write $x\ \mathsf{F}_N\ y$ for $(x, y) \in \mathsf{F}_N$. We shall denote $S_N \cup T_N$ by N and omit subscripts when no confusion can arise. As usual, the *pre-* and *postset* of $x \in N$, are ${}^\bullet x = \{y \in N \mid y\ \mathsf{F}\ x\}$ and $x^\bullet = \{y \in N \mid x\ \mathsf{F}\ y\}$. We assume ${}^\bullet t \neq \emptyset$ for all $t \in T$.

A *marking* $u : S \to \mathbb{N}$ is a finite multiset of places. It can be written either as $u = \{n_1 a_1, \ldots, n_k a_k\}$ where each n_i dictates the number of *tokens* in a_i (we omit $n_i a_i$ if $n_i = 0$), or as the formal sum $u = \bigoplus_{a_i \in S} n_i a_i$ denoting an element of the free commutative monoid $S^\oplus$ on S (the monoidal operation is defined by $(\bigoplus_i n_i a_i) \oplus (\bigoplus_i m_i a_i) = (\bigoplus_i (n_i + m_i) a_i)$ with 0 as the unit). We shall overload the symbol $\subseteq$ to denote multiset inclusion.

A *marked place/transition Petri net* (PT net) is a tuple $N = (S, T, \mathsf{F}, \mathsf{W}, u_{\text{in}})$ such that (S, T, F) is a net, the function $\mathsf{W} : \mathsf{F} \to \mathbb{N}$ assigns a positive *weight* to each arc in F, and the finite multiset $u_{\text{in}} : S \to \mathbb{N}$ is the *initial marking* of N.

We find it convenient to view F as a function $\mathsf{F} : (S \times T) \cup (T \times S) \to \{0, 1\}$, with $x\ \mathsf{F}\ y \iff \mathsf{F}(x, y) \neq 0$. Then, for PT nets we can express both F and W as the *multiset relation* $\mathsf{F} : (S \times T) \cup (T \times S) \to \mathbb{N}$. For any transition $t \in T$, let $\mathrm{pre}(t)$ and $\mathrm{post}(t)$ be the multisets over S such that $\mathrm{pre}(t)(a) = \mathsf{F}(a, t)$ and $\mathrm{post}(t)(a) = \mathsf{F}(t, a)$, for all $a \in S$. We write $t : u \to v$ for a transition t with $\mathrm{pre}(t) = u$ and $\mathrm{post}(t) = v$.

The ordinary step semantics for a PT net N is defined by the inference rules in Figure 1, assessing that in general a *step* $u \Longrightarrow_N v$ is the parallel composition (third rule) of idle resources (first rule) and transitions of the net (second rule).

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

$$\frac{u \in S^{\oplus}}{u \Longrightarrow_N u} \qquad \frac{t : u \to v \in T}{u \Longrightarrow_N v} \qquad \frac{u_1 \Longrightarrow_N v_1,\ u_2 \Longrightarrow_N v_2}{u_1 \oplus u_2 \Longrightarrow_N v_1 \oplus v_2}$$

Figure 1. The inference rules for $\Longrightarrow_N$.

$$\frac{u \in S^{\oplus}}{u \overset{\lfloor u \rfloor}{\Longrightarrow}_R u} \qquad \frac{t : u \overset{w}{\longrightarrow} v \in T}{u \oplus w \overset{w}{\Longrightarrow}_R v \oplus w} \qquad \frac{u_1 \oplus w_1 \overset{w_1}{\Longrightarrow}_R v_1 \oplus w_1,\ u_2 \oplus w_2 \overset{w_2}{\Longrightarrow}_R v_2 \oplus w_2}{u_1 \oplus u_2 \oplus \lfloor w_1 \oplus w_2 \rfloor \overset{\lfloor w_1 \oplus w_2 \rfloor}{\Longrightarrow}_R v_1 \oplus v_2 \oplus \lfloor w_1 \oplus w_2 \rfloor}$$

Figure 2. The inference rules for $\Longrightarrow_R$.

2.1. *Read arcs*

A marked *net with read arcs* (*r-net*) is a tuple $R = (S, T, \mathsf{F}, \mathsf{R}, u_{\text{in}})$, where $N_R = (S, T, \mathsf{F}, u_{\text{in}})$ is the underlying PT net and $\mathsf{R} : S \times T \to \{0, 1\}$ is the *context relation*. Though there are no technical difficulties in dealing with context multirelations (see, for example, Bruni and Montanari (2001a)), for the current presentation it is simpler to consider just context relations, bearing in mind the *maximum sharing hypothesis* of Bruni and Sassone (2000). We use $\mathrm{ctx}(t)$ to denote the (multi)set of places defined by $\mathrm{ctx}(t)(a) = \mathsf{R}(a, t)$ for all $a \in S$, and $\lfloor u \rfloor$ to denote the set $\{a \mid u(a) > 0\}$ of non-empty places of u. The minimum amount of resources that t requires to be enabled is $\mathrm{pre}(t) \oplus \mathrm{ctx}(t)$: the tokens in $\mathrm{pre}(t)$ are fetched, while those in $\mathrm{ctx}(t)$ are just read, and other transitions can access them, concurrently with t. For $t \in T$ with $\mathrm{pre}(t) = u$, $\mathrm{post}(t) = v$ and $\mathrm{ctx}(t) = w$, we write $t : u \overset{w}{\longrightarrow} v$.

For X a multiset of transitions, and u a marking, we say that X is enabled at u if $\lfloor \bigoplus_{t \in X} \mathrm{ctx}(t) \rfloor \oplus \bigoplus_{t \in T} X(t) \cdot \mathrm{pre}(t) \subseteq u$. We say that u evolves to the marking v *via* X, written $u\ [X\rangle\ v$, if X is enabled at u and $v = u \oplus \bigoplus_{t \in T} X(t) \cdot \mathrm{post}(t) \ominus \bigoplus_{t \in T} X(t) \cdot \mathrm{pre}(t)$, with $\ominus$ denoting multiset difference. Note that if u has enough tokens to satisfy the 'context' of X as well, then v is obtained from u just by removing $\bigoplus_{t \in T} X(t) \cdot \mathrm{pre}(t)$ and then adding $\bigoplus_{t \in T} X(t) \cdot \mathrm{post}(t)$.

The step relation can be defined equivalently by the inference rules in Figure 2, which also carry information about the context used in the step. The meaning of $u \overset{w}{\Longrightarrow}_R v$ is that a step can lead from u to v reading w (note that $\lfloor u \rfloor \supseteq w \subseteq \lfloor v \rfloor$). Idle tokens contribute to contexts (first rule). Transitions yield basic steps (second rule). When building larger steps, the maximum common context of the substeps is shared (third rule).

2.2. *Inhibitor arcs*

We recall that an *ri-net* is an r-net $(S, T, \mathsf{F}, \mathsf{R}, u_{\text{in}})$ together with a relation $\mathsf{I} \subseteq T \times S$, which expresses negative enabling conditions: if $\mathsf{I}(t, a)$, then t is never enabled at u with $u(a) > 0$. We let $t^{\neg} = \{a \mid \mathsf{I}(t, a)\}$, and assume that relations F, R, I are pairwise disjoint. We use the notation $t : X \multimap u \overset{w}{\longrightarrow} v$ for a transition t such that $\mathrm{pre}(t) = u$, $\mathrm{post}(t) = v$, $\mathrm{ctx}(t) = w$ and $t^{\neg} = X$.

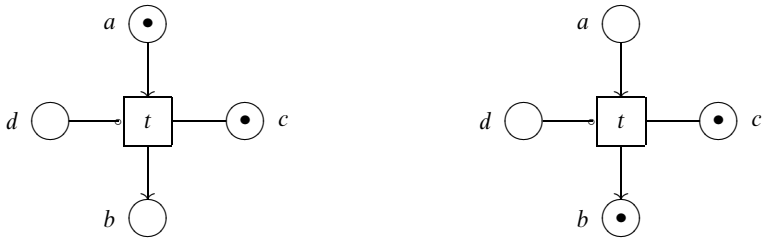
The operational semantics of ordinary ri-nets C is recalled in Figure 3, where in the third rule we assume the side condition $\Psi \overset{\text{def}}{\Longleftrightarrow} (X_1 \oplus X_2) \cap (u_1 \oplus u_2 \oplus w_1 \oplus w_2 \oplus v_1 \oplus v_2) = \emptyset$. In fact, when composing two steps in parallel, we must check that neither step involves tokens that inhibit the execution of the other step. A generic step $X \multimap u \overset{w}{\Longrightarrow}_C v$ means

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

$$\frac{u \in S^{\oplus}}{\emptyset \vdash\!\circ\ u \overset{\lfloor u \rfloor}{\Longrightarrow_C} u} \qquad \frac{t : X \vdash\!\circ\ u \xrightarrow{w} v \in T}{X \vdash\!\circ\ u \oplus w \overset{w}{\Longrightarrow_C} v \oplus w}$$

$$\frac{X_1 \vdash\!\circ\ u_1 \oplus w_1 \overset{w_1}{\Longrightarrow_C} v_1 \oplus w_1,\ X_2 \vdash\!\circ\ u_2 \oplus w_2 \overset{w_2}{\Longrightarrow_C} v_2 \oplus w_2}{\lfloor X_1 \oplus X_2 \rfloor \vdash\!\circ\ u_1 \oplus u_2 \oplus \lfloor w_1 \oplus w_2 \rfloor \overset{\lfloor w_1 \oplus w_2 \rfloor}{\Longrightarrow_C} v_1 \oplus v_2 \oplus \lfloor w_1 \oplus w_2 \rfloor} \quad \textit{if } \Psi$$

Figure 3. The inference rules for $\Longrightarrow_C$.



(a) The transition t is enabled in $a \oplus c$.

(b) After the step $d \vdash\!\circ\ a \oplus c \overset{c}{\Longrightarrow_C} b \oplus c$.

Figure 4. Graphical notation for nets.

that v can be reached from u reading w, and also that this step can be applied inside any other marking $u' \supseteq u$ provided that $u'(a) = 0$ for all $a \in X$. Note that for any such step we have $X \cap (u \oplus v) = \emptyset$, because $\mathsf{F} \cap \mathsf{I} = \emptyset$.

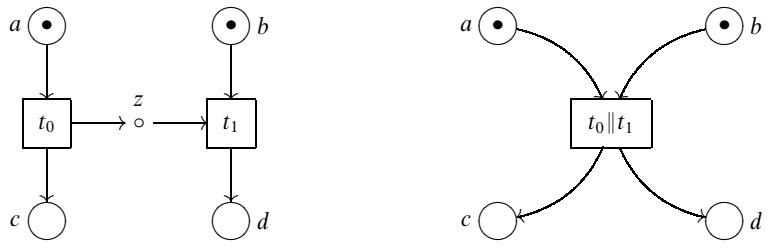
Nets allow for a friendly graphical representation, according to which places are drawn as circles, transitions as boxes, tokens as black bullets, the flow relation is represented by weighted directed arcs, read arcs are undirected, and the heads of inhibitor arcs carry a small circle (where they meet transition boxes). As an example, see the drawings in Figure 4, with $t : d \vdash\!\circ\ a \xrightarrow{c} b$.

2.3. *Zero-safe nets*

According to the ordinary terminology, in a '0-safe' net no places can contain any token in all reachable markings. We use the terminology *zero-safe net* – note the word 'zero' instead of the digit '0' – to mean that the net contains special places, called *zero places*, whose role is to coordinate the atomic execution of transitions. However, no new interaction mechanism is needed, and the coordination of the transitions participating in a step is handled by the usual token-pushing rules of nets, assuming late delivery of stable tokens (postponed to the end of the transaction). Zero places cannot contain any token in any *observable* state.

A *zero-safe net* (ZS net) is a tuple $B = (S, T, \mathsf{F}, u_{\text{in}}, Z)$ where $N_B = (S, T, \mathsf{F}, u_{\text{in}})$ is the *underlying* PT net of B and $Z \subseteq S$ is the set of *zero places*. The places in $L_B = S \setminus Z$ are called *stable places*. Note that S denotes the set of all places.

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press



(a) A zs net. (b) The abstract view.

Figure 5. Transition synchronisation on the zero place z.

$$\frac{u \oplus x \overset{w}{\Longrightarrow}_{R_D} v \oplus y}{(u,x) \overset{w}{\rightrightarrows}_D (v,y)} \qquad \frac{(u_1 \oplus w, x) \overset{w}{\rightrightarrows}_D (v_1 \oplus w, y),\ (u_2 \oplus w, y) \overset{w}{\rightrightarrows}_D (v_2 \oplus w, y')}{(u_1 \oplus u_2 \oplus w, x) \overset{w}{\rightrightarrows}_D (v_1 \oplus v_2 \oplus w, y')} \qquad \frac{(u,0) \overset{w}{\rightrightarrows}_D (v,0)}{u \overset{w}{\Rightarrow}_D v}$$

Figure 6. The inference rules for $\overset{w}{\Rightarrow}_D$.

A *stable marking* is a multiset of stable places, and the initial marking u_{in} must be stable. Stable markings describe *observable* states, whereas the presence of one or more zero tokens in a given marking makes it *unobservable*. We call *stable tokens* and *zero tokens* the tokens that belong to stable and zero places, respectively. Since $S^{\oplus}$ is a free commutative monoid, $S^{\oplus} \simeq L^{\oplus} \times Z^{\oplus}$, and we can write $t : (u,x) \to (v,y)$ for a transition t with $\text{pre}(t) = u \oplus x$ and $\text{post}(t) = v \oplus y$, where u and v are stable multisets and $x, y \in Z^{\oplus}$.

Graphically, zero places are drawn as smaller circles. For example, the net in Figure 5(a) has four stable places (a, b, c and d) and one zero place (z): in the initial marking $a \oplus b$, the firing of t_0 opens a transaction that can be closed by t_1 fetching the token in z. Note that the token in c will be released only at commit time, together with the token in d.

Zero-safe nets were introduced in Bruni and Montanari (1997; 1998), and then extended in Bruni and Montanari (2001a), by allowing the combined use of zero places and read arcs. A zs *r-net* is a tuple $D = (S, T, \mathsf{F}, \mathsf{R}, Z, u_{\text{in}})$ such that $R_D = (S, T, \mathsf{F}, \mathsf{R}, u_{\text{in}})$ is a r-net and $(S, T, \mathsf{F}, Z, u_{\text{in}})$ is an zs net. Though zero places can be used as context in Bruni and Montanari (2001a), for simplicity we shall assume that $\text{ctx}(t) \subseteq S \setminus Z$ for all $t \in T$. In defining the dynamics of zs r-nets, we can follow two main alternatives. The crucial point is whether or not to forbid a stable token to be read (possibly many times) and then also fetched during the same transaction. In the following we consider the semantics that forbids these kinds of consumption and is illustrated in Figure 6 (where $u, v, w \in L^{\oplus}$ and $x, y, y' \in Z^{\oplus}$): if a stable token is read in the transaction, then it must be preserved until commit. The second rule is crucial: it sequentialises on zero tokens, while composing in parallel on stable tokens. Note that the whole stable context w of the two substeps must be shared, because the context must be present during all the transaction steps. Idle tokens are allowed as context.

At the abstract level, the system modelled by D can be equivalently described *via* an r-net A_D such that $S_{A_D} = S_D \setminus Z_D$ and $(_ \Rightarrow_{A_D} _) = (_ \overset{w}{\Rightarrow}_D _)$, see, for example, Figure 5(b) for the abstract r-net of the zs r-net in Figure 5(a). Among the several r-nets that satisfy

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

$S ::= 0 \mid \eta.C \mid \mu?C : C$	$\eta ::= out(a) \mid rd(a) \mid in(a) \mid !in(a)$
$C ::= S \mid C\|C$	$\mu ::= rdp(a) \mid inp(a)$
$P ::= M \mid C \mid P\|P$	$M ::= \langle a \rangle$

Figure 7. Linda-like process calculus.

these conditions we can choose the optimal one, whose transitions represent the proofs of transaction steps $u \overset{w}{\Rightarrow}_D v$ taken up to equivalence (permutation of concurrent events) and that cannot be decomposed into shorter proofs. When these two conditions are satisfied, the concurrent kernel of the behaviour has been identified, and all the steps can be generated by it. (The abstract net can be defined according to either the collective or the individual token philosophy noticed in van Glabbeek and Plotkin (1995): the two approaches yield the same step relation but different abstract nets.)

3. A Linda-like process algebra

Thanks to their visual representation and their straightforward encoding of concurrency, nets have often been used as a computational model for concurrent and distributed languages such as Linda. In Busi *et al.* (1996) and Busi (1998), first a process calculus is introduced, which embeds all Linda primitives (see also Busi *et al.* (1998) and Busi *et al.* (2000a)) and then ri-nets are used to give a truly concurrent semantics to the calculus.

The processes P (also called agents) of the calculus are defined in Figure 7, where S denotes sequential programs and C denotes *concurrent programs*. Briefly, 0 is the inactive process; M represents a message available in the tuple space; prefixing can involve either the emission $out(a)$ of the tuple $\langle a \rangle$, or the fetching $in(a)$ of the tuple $\langle a \rangle$, or the reading without consuming $rd(a)$, or, also, the replicated input $!in(a)$; processes $\mu?C_1 : C_2$, with μ either the input predicate $inp(a)$ or the read predicate $rdp(a)$, are conditional constructs such that C_1 is executed if the tested message $\langle a \rangle$ is in the tuple space (in the case of $\mu = inp(a)$ the message is also removed), otherwise C_2 is executed; and $P|Q$ is the usual parallel composition. A concurrent program whose topmost operator is 0 or a prefix or a conditional statement is called a sequential program. A process can always be seen as a pair formed by a tuple space (a multiset of messages M) and a concurrent program (a multiset of sequential programs).

The distinction between processes P and concurrent programs C means that new tuples can only be added dynamically to the tuple space *via* output operations performed by some concurrent programs, though some tuples can already be present in the initial state, which is a process. For example, the expression $out(a).(\langle a \rangle|\langle a \rangle|C)$ that could perform the instantaneous output of three messages $\langle a \rangle$ and then behave like C is not allowed by the syntax in Figure 7. In the following, we shall slightly abuse the notation by writing, for example, $\eta.P$ and $\mu?P_1 : P_2$ with the implicit typing assumption that P, P_1 and P_2 denote concurrent programs and not generic processes.

The interleaving semantics of the Linda-like process algebra in Figure 7 can be given as the labelled transition system defined by the inductive rules in Figure 8, where α

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

$$\frac{}{\langle a\rangle \xrightarrow{[\overline{a}]} \langle a\rangle}\ (\overline{read}) \qquad \frac{}{\langle a\rangle \xrightarrow{\overline{a}} 0}\ (\overline{in}) \qquad \frac{}{out(a).P \xrightarrow{\tau} \langle a\rangle|P}\ (out)$$

$$\frac{}{rd(a).P \xrightarrow{[a]} P}\ (read) \qquad \frac{}{in(a).P \xrightarrow{a} P}\ (in) \qquad \frac{}{!in(a).P \xrightarrow{a} P|!in(a).P}\ (rep)$$

$$\frac{}{rdp(a)?P : Q \xrightarrow{[a]} P}\ (readp+) \qquad \frac{}{inp(a)?P : Q \xrightarrow{a} P}\ (inp+) \qquad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}\ (Lpar)$$

$$\frac{}{rdp(a)?P : Q \xrightarrow{\neg a} Q}\ (readp-) \qquad \frac{}{inp(a)?P : Q \xrightarrow{\neg a} Q}\ (inp-) \qquad \frac{P \xrightarrow{\alpha} P'}{Q|P \xrightarrow{\alpha} Q|P'}\ (Rpar)$$

$$\frac{P \xrightarrow{[a]} P',\ Q \xrightarrow{[\overline{a}]} Q'}{P|Q \xrightarrow{\tau} P'|Q'}\ (LreadOK) \qquad \frac{P \xrightarrow{a} P',\ Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}\ (LinOK) \qquad \frac{P \xrightarrow{\neg a} P',\ Q \not\xrightarrow{\overline{a}}}{P|Q \xrightarrow{\neg a} P'|Q}\ (LcheckOK)$$

Figure 8. Interleaving (ordered) semantics of Linda-like process calculus.

ranges over $\{[\overline{a}], \overline{a}, [a], a, \tau\}$ and the obvious right-symmetric rules (*RreadOK*), (*RinOK*) and (*RcheckOK*) are omitted. We shall use the term *environment* to denote the multiset of messages and sequential programs that possibly surrounds a given process P. Transitions are labelled with six different kinds of observations:

1 $P \xrightarrow{[\overline{a}]} P'$ means that $P' = P$ and that the message $\langle a\rangle$ is available for reading in the tuple space of P;
2 $P \xrightarrow{\overline{a}} P'$ means that the message $\langle a\rangle$ is available for fetching in the tuple space of P and that P' is the residual of P after the fetching of $\langle a\rangle$;
3 $P \xrightarrow{[a]} P'$ means that P can evolve to P' after the reading of the message $\langle a\rangle$ from the environment;
4 $P \xrightarrow{a} P'$ means that P can evolve to P' after the fetching of the message $\langle a\rangle$ from the environment;
5 $P \xrightarrow{\tau} P'$ means that P can evolve to P' without any interaction with the environment;
6 $P \xrightarrow{\neg a} P'$ means that P can evolve to P' only if (the tuple space of) the environment does not contain the message $\langle a\rangle$.

Due to rule $(\overline{in})$, the rule $(\overline{read})$ and the observation $[\overline{a}]$ are redundant. They could be safely removed by changing the rules (*LreadOK*) and (*RreadOK*) as below (*cf.* Busi *et al.* (1998)):

$$\frac{P \xrightarrow{[a]} P',\ Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q}\ (LreadOK) \qquad \frac{P \xrightarrow{[a]} P',\ Q \xrightarrow{\overline{a}} Q'}{Q|P \xrightarrow{\tau} Q|P'}\ (RreadOK)$$

In fact, using Q in the target of the conclusions has the same effect as putting back the message $\langle a\rangle$. However, we prefer to keep the transition system in Figure 8, which better reflects the semantics of reading without consumption. Note also that the negative premise in rules (*LcheckOK*) and (*RcheckOK*) does not introduce inconsistency because

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

$$\frac{}{out(a).P \xrightarrow{\tau} \langle\langle a\rangle\rangle | P} \ (emit) \qquad \frac{}{\langle\langle a\rangle\rangle \xrightarrow{\tau} \langle a\rangle} \ (rend)$$

Figure 9. Unordered semantics.

$$\begin{aligned} dec(0) &\stackrel{\text{def}}{=} 0 \\ dec(M) &\stackrel{\text{def}}{=} M \\ dec(\langle M\rangle) &\stackrel{\text{def}}{=} \langle M\rangle \\ dec(\eta.P) &\stackrel{\text{def}}{=} \eta.P \\ dec(\mu?P : Q) &\stackrel{\text{def}}{=} \mu?P : Q \\ dec(P|Q) &\stackrel{\text{def}}{=} dec(P) \oplus dec(Q) \end{aligned}$$

Figure 10. Decomposition function.

the system is strictly stratifiable (Groote 1993) (essentially, the test $Q \stackrel{\overline{a}}{\not\longrightarrow}$ is a structural predicate on Q that checks for the presence/absence of $\langle a\rangle$).

It should be evident that the only moves that P is allowed to perform as a stand-alone entity are those labelled with τ and $\neg a$. Thus we write $P \longrightarrow P'$ if $P \xrightarrow{\tau} P'$ or $P \xrightarrow{\neg a} P'$ for some a.

It is worth noticing that the semantics of the $out(a)$ operation that we have presented is the so-called *ordered* semantics, where the emission and the rendering of a message form a single and atomic action. Alternatively, we can extend the syntax of the calculus with messages $\langle\langle a\rangle\rangle$ and let the rule (*out*) be replaced by the two rules in Figure 9 yielding the *unordered* semantics, where emission and rendering of messages are two autonomous actions. In fact, note that the particle $\langle\langle a\rangle\rangle$ cannot interact with the environment until the rule (*rend*) is applied, which can happen much later than its emission. The difference between ordered and unordered semantics is crucial, because deadlock is decidable only in the second case (it is possible to find an ordinary PT net that is deadlock equivalent to the ri-net of the original encoding (Busi *et al.* 2000b)).

3.1. *Net semantics*

The ordinary interleaving semantics does not provide a faithful account of the concurrent activities that are intuitively demanded for the language, like multiple concurrent readings of the same tuple. To this end, net semantics can be exploited by decomposing processes into multisets of basic sequential components that can be thought of as running in parallel. Net transitions must then express the local interactions between such components. In the proposed semantics of Busi *et al.* (1996) and Busi (1998), ri-nets are employed so that the $rd(a)$ operation is modelled *via* read arcs (allowing for multiple concurrent readings of a), while inhibitor arcs model the branching of conditional statements in case messages are absent.

The decomposition function (for both the ordered and unordered semantics) is defined in Figure 10. Roughly speaking, the net modelling of Linda is defined by associating

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

$$
\begin{array}{rrl}
\texttt{rd(a,P)}: & 0 \multimap & rd(a).P \xrightarrow{\langle a \rangle} \mathrm{dec}(P) \\
\texttt{rdp}+\texttt{(a,P,Q)}: & 0 \multimap & rdp(a)?P : Q \xrightarrow{\langle a \rangle} \mathrm{dec}(P) \\
\texttt{rdp}-\texttt{(a,P,Q)}: & \langle a \rangle \multimap & rdp(a)?P : Q \xrightarrow{0} \mathrm{dec}(Q) \\
\texttt{in(a,P)}: & 0 \multimap & \langle a \rangle \oplus in(a).P \xrightarrow{0} \mathrm{dec}(P) \\
\texttt{inp}+\texttt{(a,P,Q)}: & 0 \multimap & \langle a \rangle \oplus inp(a)?P : Q \xrightarrow{0} \mathrm{dec}(P) \\
\texttt{inp}-\texttt{(a,P,Q)}: & \langle a \rangle \multimap & inp(a)?P : Q \xrightarrow{0} \mathrm{dec}(Q) \\
\texttt{!in(a,P)}: & 0 \multimap & \langle a \rangle \xrightarrow{!in(a).P} \mathrm{dec}(P) \\
\texttt{out(a,P)}: & 0 \multimap & out(a).P \xrightarrow{0} \langle a \rangle \oplus \mathrm{dec}(P) \\
 & & \\
\texttt{emit(a,P)}: & 0 \multimap & out(a).P \xrightarrow{0} \langle\langle a \rangle\rangle \oplus \mathrm{dec}(P) \\
\texttt{rend(a)}: & 0 \multimap & \langle\langle a \rangle\rangle \xrightarrow{0} \langle a \rangle
\end{array}
$$

Figure 11. Basic transitions for the net semantics of Linda.

distinct places with messages $\langle a \rangle$ and sequential processes P, and suitable transitions with computational moves of sequential processes. A marking of the resulting net is a multiset of active sequential processes and messages in the tuple space.

The transitions expressing reductions between sequential processes and messages in the tuple space are given in Figure 11, where the sets of transitions $\{\texttt{out(a,P)}\}$ and $\{\texttt{emit(a,P)}, \texttt{rend(a)}\}$ are mutually exclusive and characterise the ordered and unordered semantics, respectively. Note that the only transitions with non-empty negative premises are $\texttt{rdp}-\texttt{(a,P,Q)}$ and $\texttt{inp}-\texttt{(a,P,Q)}$, while read arcs are required for $\texttt{rd(a,P)}$, $\texttt{rdp}+\texttt{(a,P,Q)}$ and, also, $\texttt{!in(a,P)}$.

The net for the whole language is of course infinite. However, if the initial marking is $\mathrm{dec}(P)$ for P a process, then the sets of fireable transitions and of markable places are finite and can be approximated by restricting to the finite ri-net $net(P)$:

— whose set of places is the union between the set of sequential subprocesses Q of P and the set of messages $\langle a \rangle$ (and $\langle\langle a \rangle\rangle$ for the unordered semantics) for any message name a that appears in P;
— whose set of transitions consists of all transitions acting on these places; and
— whose initial marking is $\mathrm{dec}(P)$.

The correspondence between the interleaving semantics and the net semantics can be drawn as follows.

Proposition 3.1 (*cf.* Busi *et al.* (1996)). Let P be a process.

— If $P \longrightarrow P_1 \cdots \longrightarrow P_n$, then $\mathrm{dec}(P) \rightarrow_{net(P)} \mathrm{dec}(P_1) \cdots \rightarrow_{net(P)} \mathrm{dec}(P_n)$.
— If $\mathrm{dec}(P) \rightarrow_{net(P)} u_1 \cdots \rightarrow_{net(P)} u_n$, then $P_1, \ldots, P_n$ exist such that $P \longrightarrow^* P_1 \cdots \longrightarrow^* P_n$ and $\mathrm{dec}(P_i) = u_i$ for all $i \in [1, n]$.

Where the notation $u \rightarrow_{net(P)} v$ means that there exist a negative context X and a positive context w (with $X \cap (u \oplus v) = \emptyset$ and $u \supseteq w \subseteq v$) such that $X \multimap u \stackrel{w}{\Longrightarrow}_{net(P)} v$.

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

Note that, in general, a step of $net(P)$ can involve several firings and thus must be simulated by a sequence of moves in the interleaving semantics.

Remark 3.2. For simplicity, the non-deterministic choice operator $P + Q$ is not considered here. However, it can be dealt with using either the classical approach for distributed choices (involving the cartesian product of dec(P) and dec(Q)), or the technique presented in Bruni and Montanari (2001a).

4. Adding transactions to Linda

The different expressiveness between the ordered and unordered semantics of Linda witnesses that atomicity can play an important role in the design of coordination languages. Moreover, this consideration applies not only to the primitives of the language, but also to certain applications that users must be able to define. For example, most commercial applications are based on transactions for which the so-called ACID properties (atomicity, consistency, isolation and durability) must be guaranteed, at least in some relaxed form.

Different primitives for transactions are considered, for example, in JavaSpaces, TSpaces and under .NET's BizTalk Orchestration language, which are all based on the idea of grouping series of actions into atomic blocks. The zero-safe approach is based instead on a more dynamical notion of grouping, which is determined by the flow of data (tokens) between threads, so that many entities can start their own transactions independently and then, by exchanging data and information, can join each other's transactions. The coordination of all threads belonging to the same transaction is then crucial, because they must all commit or abort in case the transaction fails.

Given the net modelling of Linda described in the previous section, the idea is to design the transaction primitives to be added to Linda taking inspiration from the zero-safe approach. However, to do so, we must first consider the possibility of adding inhibitor arcs to zero-safe nets, which are needed for modelling conditional constructs, and we shall see that some limitations must be imposed to have a consistent abstract view of the system.

A comparison with other approaches in the literature is given in Section 5.

4.1. *Zero-safe nets with inhibitor arcs*

Our first goal is to extend the zero-safe approach to nets with inhibitor arcs. This is not completely straightforward, as the following example demonstrates.

Example 1. Let us consider the net in Figure 12. At the abstract level, there are two apparently independent transactions: one from a_1 to a_2 (atomic execution of t_1 followed by t_2) and one from a_3 to a_4, but of course at the low level these activities can hardly be seen as independent because of the token flow in z_1, which disappears at the abstract level. On the other hand, viewing t_3 as part of the same transaction as t_1 and t_2 is again not correct, because it can also fire in isolation.

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

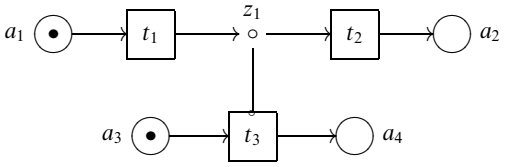


Figure 12. Is the abstract behaviour of this net correct?

$$\frac{X \vdash\!\circ\ u \oplus x \overset{w}{\Longrightarrow}_{C_E} v \oplus y}{X \vdash\!\circ\ (u,x) \overset{w}{\rightrightarrows}_E (v,y)} \qquad \frac{X \vdash\!\circ\ (u,0) \overset{w}{\rightrightarrows}_E (v,0)}{X \vdash\!\circ\ u \overset{w}{\Rightarrow}_E v}$$

$$\frac{X_1 \vdash\!\circ\ (u_1 \oplus w, x) \overset{w}{\rightrightarrows}_E (v_1 \oplus w, y),\ X_2 \vdash\!\circ\ (u_2 \oplus w, y) \overset{w}{\rightrightarrows}_E (v_2 \oplus w, y')}{\lfloor X_1 \oplus X_2 \rfloor \vdash\!\circ\ (u_1 \oplus u_2 \oplus w, x) \overset{w}{\rightrightarrows}_E (v_1 \oplus v_2 \oplus w, y')}\ \textit{if}\ \Psi$$

Figure 13. The inference rules for $\Rightarrow_E$.

The problem is that negative conditions due to testing for absence of zero tokens are no longer representable at the abstract level, where zero tokens cannot be viewed. To solve this ambiguity, we restrict the usage of inhibitor arcs to stable places only.

Definition 4.1. A *zero-safe ri-net* (zs ri-net) is a tuple $E = (S, T, \mathsf{F}, \mathsf{R}, \mathsf{I}, u_{\text{in}}, Z)$, where $D_E = (S, T, \mathsf{F}, \mathsf{R}, u_{\text{in}}, Z)$ is a zs r-net, and $\mathsf{I} \subseteq T \times (S \setminus Z)$ is the *inhibitor relation*. The underlying ri-net of E is $C_E = (S, T, \mathsf{F}, \mathsf{R}, \mathsf{I}, u_{\text{in}})$

The operational semantics of zs ri-nets is then the obvious extension illustrated in Figure 13 (where in the third rule we again assume the side condition $\Psi \overset{\text{def}}{\Longleftrightarrow} (X_1 \oplus X_2) \cap (u_1 \oplus u_2 \oplus w \oplus v_1 \oplus v_2) = \emptyset$). Note that the concatenation rule composes stable places in parallel and zero places in series. Since the only places that can be attached to inhibitor arcs are stable, it follows that a transaction never serialises tokens in places tested for absence.

Analogously to Bruni and Montanari (2000b), we can define the abstract ri-net A_E of E according to the two main philosophies of concurrency (collective or individual token), by characterising the minimal concurrent steps $X \vdash\!\circ\ u \overset{w}{\Rightarrow}_E v$ and viewing them as atomic activities with preset u, postset v, reading w and inhibited by X. We shall discuss in Section 6 a convenient way of expressing this construction in the case of zs r-nets.

4.2. *Transactions in TraLinda*

Our second goal is to exploit zs ri-nets as foundational models of Linda-like languages extended with transactions. Given the modelling in Busi *et al.* (1996) and Busi (1998), we know that places can be conveniently used to represent either sequential processes or messages; hence, the idea is to distinguish between *low-level* (that is, zero-safe) and *high-level* (that is, stable) entities.

For messages we rely on two disjoint types, $\mathtt{H}$ and $\mathtt{L}$, and a typing relation $a : \tau$ with $\tau \in \{\mathtt{H}, \mathtt{L}\}$. Since in zs ri-net models zero places cannot be used as contexts, the prefixes η

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

$S ::= 0 \mid \eta.C \mid \eta_ C \mid \mu?C : C$	$\eta ::= out(a : \tau) \mid rd(a : \mathtt{H}) \mid in(a : \tau) \mid !in(a : \tau)$
$C ::= S \mid C \mid C$	$\mu ::= rdp(a : \mathtt{H}) \mid inp(a : \mathtt{H})$
$P ::= M \mid C \mid P \mid P$	$M ::= \langle a : \tau \rangle$

Figure 14. TraLinda.

and the predicates μ of conditional expressions can only read and test for the presence of messages of the form $a : \mathtt{H}$. This restriction is arguable, but it is necessary for defining the abstract net models of processes, where zero-safe messages are not visible, as discussed in Example 1.

Regarding sequential components, we find it convenient to introduce the atomic prefixing $\eta_ C$, with the meaning that the successful execution of η is subordinated to the successful execution of the next activities of C (that is, the commit of C). For example, the process $in(a : \mathtt{H})_ (out(b : \mathtt{H}).0 | in(c : \mathtt{H}).0)$ can input the message $\langle a : \mathtt{H} \rangle$ provided that it also outputs the message $\langle b : \mathtt{H} \rangle$ and inputs $\langle c : \mathtt{H} \rangle$.

The resulting language, which we call TraLinda, is in Figure 14.

The semantic framework provided by zs ri-nets can be exploited first by taking the finite ri-net $net(P)$ for each agent P, and then by considering the subset of places associated to temporary messages as the set of zero places. To handle atomic prefixing, we also introduce a zero place $\widehat{P}$ for each sequential agent P, which replaces P in the postsets of transitions associated to (atomic) prefixing. However, two special cases must be considered:

(1) the inactive process, for which having a zero place $\widehat{0}$ would mean always abort the transactions that put a token in it (since that token can never be removed);
(2) the guarded replication prefix, because the agent $!in(a : \tau).P$ is always accessed in reading, and, again, tokens will never be fetched from $!in(\widehat{a : \tau}).P$.

Instead of leaving to the user's skills the possibility of exploiting these places for special purposes (for example, aborting the transaction), we assume that $\widehat{0}$, $!in(\widehat{a : \tau}).P$ and $!in(\widehat{a : \tau})_ P$ are stable and coincide with 0, $!in(a : \tau).P$, and $!in(a : \tau)_ P$, respectively. In particular, we have, for example, that $\eta_ 0$ is completely equivalent to $\eta.0$, and that $\eta_ !in(a : \tau).P$ is equivalent to $\eta.!in(a : \tau).P$.

Formally, we let the decomposition function be defined in Figure 15, while the basic transitions are in Figure 16, where for $\mathrm{dec}(P) = \{P_1, \ldots, P_n, M_1, \ldots, M_m\}$ with P_i's being sequential processes and M_j's being messages, we have used the notation $\widehat{\mathrm{dec}(P)}$ to denote the multiset $\{\widehat{P_1}, \ldots, \widehat{P_n}, M_1, \ldots, M_m\}$. Briefly, the places $\langle a : \mathtt{H} \rangle$ are stable, while places $\langle a : \mathtt{L} \rangle$ are zero-safe. Transitions for the reduction of prefixes are available in several variants:

1 the ordinary version, which is analogous to the definition in Figure 11, and typically consumes a token from $\eta.P$ and produces the multiset of tokens $\mathrm{dec}(P)$ (for example, $\mathtt{rd(a,P)}$);
2 the 'hat' version, which takes a zero token $\widehat{\eta.P}$ and again produces $\mathrm{dec}(P)$ (for example, $\widehat{\mathtt{rd(a,P)}}$);
3 the atomic-prefix variant, which fetches from $\eta_ P$ and produces the multiset of tokens $\widehat{\mathrm{dec}(P)}$, where all released sequential components are zero-safe (for example, $\mathtt{atrd(a,P)}$);

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

$$\begin{aligned}
\mathrm{dec}(0) &\stackrel{\mathrm{def}}{=} 0\\
\mathrm{dec}(M) &\stackrel{\mathrm{def}}{=} M\\
\mathrm{dec}(\langle M\rangle) &\stackrel{\mathrm{def}}{=} \langle M\rangle\\
\mathrm{dec}(\eta.P) &\stackrel{\mathrm{def}}{=} \eta.P\\
\mathrm{dec}(\eta_\ P) &\stackrel{\mathrm{def}}{=} \eta_\ P\\
\mathrm{dec}(\mu?P:Q) &\stackrel{\mathrm{def}}{=} \mu?P:Q\\
\mathrm{dec}(P|Q) &\stackrel{\mathrm{def}}{=} \mathrm{dec}(P)\oplus\mathrm{dec}(Q)
\end{aligned}$$

Figure 15. Decomposition function.

4 the hat version of the atomic-prefix variant, which fetches from $\widehat{\eta_\ P}$ and produces $\widehat{\mathrm{dec}(P)}$ (for example, $\widehat{\texttt{atrd(a,P)}}$).

Analogously, for each conditional statement we have four variants: one for the ordinary successful test; one for the ordinary negative test; and the hat versions of each of these. Note that the tokens produced by the hat versions are zero-safe, unless the continuation P (or Q) is 0 or a replicated input, in which case $\widehat{\mathrm{dec}(P)} = \mathrm{dec}(P)$.

In the case of guarded replication, hat versions are unnecessary, because we have assumed that $!in(\widehat{a:\tau}).P = !in(a:\tau).P$ and $!in(\widehat{a:\tau})_\ P = !in(a:\tau)_\ P$, and that these places are stable. Note that this fact has some consequences in the encoding of more complex terms like $\eta_\ !in(a:\tau).P$, where the atomic prefix behaves as the ordinary prefix (because $!in(\widehat{a:\tau}).P$ is stable). Conditional constructs have no variant and can only test stable places for absence/presence of tokens.

The transitions `emit` and `rend` (in all their flavours) are not considered in the ordered semantics and replace transitions `out` in the unordered semantics. In the following, unless otherwise specified, we consider the ordered semantics.

For each process P, the resulting net $E(P)$ obtained by considering only sequential subcomponents of P and messages in P is a finite zs ri-net, and, as such, it comes equipped with an abstract view of the system, which is the ri-net $A_{E(P)}$.

Definition 4.2. Given a process P, the zs ri-net $E(P)$ is constructed as follows:

1 Let $S_0 = \{Q \mid Q \in \mathrm{dec}(P)\}$, $T_0 = \varnothing$ and $i = 0$.
2 Let $T_{i+1} = T_i \cup T_i'$, where T_i' is the set of transitions in Figure 16 whose preset or context contain a sequential component that is in S_i.
3 Let $S_{i+1} = S_i \cup S_i'$ where S_i' is the set of places involved in any of the transitions in T_i'.
4 If $S_{i+1} = S_i$ and $T_{i+1} = T_i$ stop, otherwise increment i and iterate on step (2).

Proposition 4.3. Given a process P, the zs ri-net $E(P)$ is computed in a finite number of steps.

Proof. The sets of places and transitions of $E(P)$ are trivially bounded by the sets of subcomponents of P and of transitions acting on them, respectively. □

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

$$
\begin{array}{rll}
\mathtt{rd(a,P)}: & 0 \multimap & rd(a:\mathtt{H}).P \xrightarrow{\langle a:\mathtt{H}\rangle} \mathrm{dec}(P) \\
\widehat{\mathtt{rd(a,P)}}: & 0 \multimap & rd(\widehat{a:\mathtt{H}}).P \xrightarrow{\langle a:\mathtt{H}\rangle} \mathrm{dec}(P) \\
\mathtt{atrd(a,P)}: & 0 \multimap & rd(a:\mathtt{H})_\ P \xrightarrow{\langle a:\mathtt{H}\rangle} \widehat{\mathrm{dec}(P)} \\
\widehat{\mathtt{atrd(a,P)}}: & 0 \multimap & rd(\widehat{a:\mathtt{H}})_\ P \xrightarrow{\langle a:\mathtt{H}\rangle} \widehat{\mathrm{dec}(P)} \\
\mathtt{rdp+(a,P,Q)}: & 0 \multimap & rdp(a:\mathtt{H})?P\ :Q \xrightarrow{\langle a:\mathtt{H}\rangle} \mathrm{dec}(P) \\
\widehat{\mathtt{rdp+(a,P,Q)}}: & 0 \multimap & rdp(\widehat{a:\mathtt{H}})?P\ :Q \xrightarrow{\langle a:\mathtt{H}\rangle} \widehat{\mathrm{dec}(P)} \\
\mathtt{rdp-(a,P,Q)}: & \langle a:\mathtt{H}\rangle \multimap & rdp(a:\mathtt{H})?P\ :Q \xrightarrow{0} \mathrm{dec}(Q) \\
\widehat{\mathtt{rdp-(a,P,Q)}}: & \langle a:\mathtt{H}\rangle \multimap & rdp(\widehat{a:\mathtt{H}})?P\ :Q \xrightarrow{0} \widehat{\mathrm{dec}(Q)} \\
\mathtt{in(a,\tau,P)}: & 0 \multimap & \langle a:\tau\rangle \oplus in(a:\tau).P \xrightarrow{0} \mathrm{dec}(P) \\
\widehat{\mathtt{in(a,\tau,P)}}: & 0 \multimap & \langle a:\tau\rangle \oplus in(\widehat{a:\tau}).P \xrightarrow{0} \mathrm{dec}(P) \\
\mathtt{atin(a,\tau,P)}: & 0 \multimap & \langle a:\tau\rangle \oplus in(a:\tau)_\ P \xrightarrow{0} \widehat{\mathrm{dec}(P)} \\
\widehat{\mathtt{atin(a,\tau,P)}}: & 0 \multimap & \langle a:\tau\rangle \oplus in(\widehat{a:\tau})_\ P \xrightarrow{0} \widehat{\mathrm{dec}(P)} \\
\mathtt{inp+(a,P,Q)}: & 0 \multimap & \langle a:\mathtt{H}\rangle \oplus inp(a:\mathtt{H})?P\ :Q \xrightarrow{0} \mathrm{dec}(P) \\
\widehat{\mathtt{inp+(a,P,Q)}}: & 0 \multimap & \langle a:\mathtt{H}\rangle \oplus inp(\widehat{a:\mathtt{H}})?P\ :Q \xrightarrow{0} \widehat{\mathrm{dec}(P)} \\
\mathtt{inp-(a,P,Q)}: & \langle a:\mathtt{H}\rangle \multimap & inp(a:\mathtt{H})?P\ :Q \xrightarrow{0} \mathrm{dec}(Q) \\
\widehat{\mathtt{inp-(a,P,Q)}}: & \langle a:\mathtt{H}\rangle \multimap & inp(\widehat{a:\mathtt{H}})?P\ :Q \xrightarrow{0} \widehat{\mathrm{dec}(Q)} \\
\mathtt{!in(a,\tau,P)}: & 0 \multimap & \langle a:\tau\rangle \xrightarrow{!in(a:\tau).P} \mathrm{dec}(P) \\
\mathtt{at!in(a,\tau,P)}: & 0 \multimap & \langle a:\tau\rangle \xrightarrow{!in(a:\tau)_\ P} \widehat{\mathrm{dec}(P)} \\
\mathtt{out(a,\tau,P)}: & 0 \multimap & out(a:\tau).P \xrightarrow{0} \langle a:\tau\rangle \oplus \mathrm{dec}(P) \\
\widehat{\mathtt{out(a,\tau,P)}}: & 0 \multimap & out(\widehat{a:\tau}).P \xrightarrow{0} \langle a:\tau\rangle \oplus \mathrm{dec}(P) \\
\mathtt{atout(a,\tau,P)}: & 0 \multimap & out(a:\tau)_\ P \xrightarrow{0} \langle a:\tau\rangle \oplus \widehat{\mathrm{dec}(P)} \\
\widehat{\mathtt{atout(a,\tau,P)}}: & 0 \multimap & out(\widehat{a:\tau})_\ P \xrightarrow{0} \langle a:\tau\rangle \oplus \widehat{\mathrm{dec}(P)} \\
 & & \\
\mathtt{emit(a,\tau,P)}: & 0 \multimap & out(a:\tau).P \xrightarrow{0} \langle\langle a:\tau\rangle\rangle \oplus \mathrm{dec}(P) \\
\widehat{\mathtt{emit(a,\tau,P)}}: & 0 \multimap & out(\widehat{a:\tau}).P \xrightarrow{0} \langle\langle a:\tau\rangle\rangle \oplus \mathrm{dec}(P) \\
\mathtt{atemit(a,\tau,P)}: & 0 \multimap & out(a:\tau)_\ P \xrightarrow{0} \langle\langle a:\tau\rangle\rangle \oplus \widehat{\mathrm{dec}(P)} \\
\widehat{\mathtt{atemit(a,\tau,P)}}: & 0 \multimap & out(\widehat{a:\tau})_\ P \xrightarrow{0} \langle\langle a:\tau\rangle\rangle \oplus \widehat{\mathrm{dec}(P)} \\
\mathtt{rend(a,\tau)}: & 0 \multimap & \langle\langle a:\tau\rangle\rangle \xrightarrow{0} \langle a:\tau\rangle
\end{array}
$$

Figure 16. Basic transitions for the net semantics of TraLinda.

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

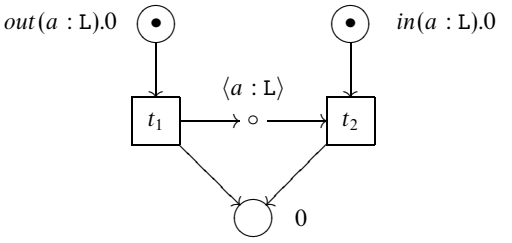


Figure 17. The zs ri-net for the process $out(a : \mathtt{L}).0|in(a : \mathtt{L}).0$.

The presence of atomic prefixing is essential for the introduction of forking transitions from one zero-safe place to two zero-safe places. If we just consider the sublanguage of TraLinda without atomic prefixing, the only admissible transactions are given by the coordination of exactly one output and one input action.

Lemma 4.4. Given a process P without atomic prefixing, the only zero-safe places in the zs ri-net $E(P)$ are those associated with zero-safe messages $\langle a : \mathtt{L}\rangle$.

Proposition 4.5. Given a process P without atomic prefixing, all the transactions of the zs ri-net $E(P)$ contains at most two transition firings.

Proof. By Lemma 4.4, transactions can only exploit zero-safe tokens of the form $\langle a : \mathtt{L}\rangle$. By a simple case analysis of transitions in Figure 16, it follows that the only transitions that can be involved in transactions are of the form $\mathtt{in}(\mathtt{a},\mathtt{L},\mathtt{Q})$, $!\mathtt{in}(\mathtt{a},\mathtt{L},\mathtt{Q})$ and $\mathtt{out}(\mathtt{a},\mathtt{L},\mathtt{Q})$. In particular, the only one that can initiate a transaction is $\mathtt{out}(\mathtt{a},\mathtt{L},\mathtt{Q})$. But then, the firing of either $\mathtt{in}(\mathtt{a},\mathtt{L},\mathtt{Q})$ or $!\mathtt{in}(\mathtt{a},\mathtt{L},\mathtt{P})$ will close the transaction. □

Note that Linda is trivially embedded in TraLinda by just considering messages of type H and avoiding atomic prefixing.

Proposition 4.6. If P is a Linda-like process, then $net(P) = E(P)$.

Proof. The proof is by induction on the way the two nets are built. □

Example 2. In $P = out(a : \mathtt{H}).0|in(a : \mathtt{H}).0$, the two sequential subprocesses $Q = out(a : \mathtt{H}).0$ and $R = in(a : \mathtt{H}).0$ can communicate asynchronously as in Linda, while in $P' = out(a : \mathtt{L}).0|in(a : \mathtt{L}).0$ a synchronisation is enforced between $Q' = out(a : \mathtt{L}).0$ and $R' = in(a : \mathtt{L}).0$ by the typing $a : \mathtt{L}$. In fact, in $E(P')$ (see Figure 17) we have the initial marking $Q' \oplus R'$, with a transition $t_1 = \mathtt{out}(a,\mathtt{L},0) : Q' \to \langle a : \mathtt{L}\rangle \oplus 0$ for the first sequential subcomponent, and a transition $t_2 = \mathtt{in}(a,\mathtt{L},0) : R' \oplus \langle a : \mathtt{L}\rangle \to 0$ for the second subprocess, where $\langle a : \mathtt{L}\rangle$ is zero-safe. Thus, the firing of t_1 opens a transaction that only the firing of t_2 can commit, resulting in the abstract transition from $Q' \oplus R'$ to $2 \cdot 0$ (two inactive processes/tokens), where the two operations are executed atomically (that is, they are synchronised at the abstract level).

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

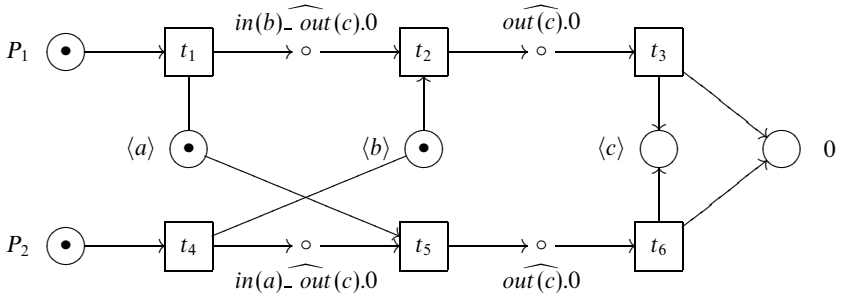


Figure 18. The zs ri-net for $P = rd(a)_\ in(b)_\ out(c).0|rd(b)_\ in(a)_\ out(c).0|\langle a\rangle|\langle b\rangle$.

Example 3. Let $a, b, c : \mathtt{H}$ and take the process

$$P = rd(a)_\ in(b)_\ out(c).0|rd(b)_\ in(a)_\ out(c).0|\langle a\rangle|\langle b\rangle.$$

where two atomic sequential threads $P_1 = rd(a)_\ in(b)_\ out(c).0$ and $P_2 = rd(b)_\ in(a)_\ out(c).0$ are in parallel with stable messages $\langle a\rangle$ and $\langle b\rangle$. The net $E(P)$ is in Figure 18. Then, only one of the two atomic threads can be executed, because the same stable token (for example, $\langle a\rangle$ or $\langle b\rangle$) cannot be first read and then consumed during the same transaction. Thus, if P_1 is executed first, the message $\langle b\rangle$ is fetched and cannot be read by P_2, while if P_2 executes, $\langle a\rangle$ is consumed. In fact, at the abstract level we have a transition from $P_1 \oplus \langle b\rangle$ to $\langle c\rangle \oplus 0$ and context $\langle a\rangle$ associated to the P_1 thread, and a transition from $P_2 \oplus \langle a\rangle$ to $\langle c\rangle \oplus 0$ and context $\langle b\rangle$ corresponding to the P_2 thread. The situation is very different if $P' = rd(a).in(b).out(c).0|rd(b).in(a).out(c).0|\langle a\rangle|\langle b\rangle$ is considered, where the two read operations can be executed concurrently first, then the two messages $\langle a\rangle$ and $\langle b\rangle$ can be retrieved, and finally two copies of the message $\langle c\rangle$ can be emitted.

Example 4. To illustrate how the transactional mechanism can be exploited to program typical transactions, let us consider an extension of TraLinda in which tuples can contain structured data, and some sort of value passing is allowed by the input (this would correspond to considering *coloured* tokens in the corresponding net). The application we want to model is a protocol for hotel room reservation. Informally, the system is composed of n hotels $P(\mathsf{H}_1), \ldots, P(\mathsf{H}_n)$ and m people $Q(\mathsf{N}_1), \ldots, Q(\mathsf{N}_m)$, where H_i is the name of the ith hotel and N_j is the name of the jth person, and all names are pairwise different. People can ask for a room on a public board Req. Hotels can pick up a request from the board and then, if a room fitting the description posted by the person is available, they can send back a price offer for that room. When a person receives an offer he/she can decide whether to confirm or decline it. The confirmation must then be registered by the hotel. If the reservation is cancelled, the room is made available again.

The generic process $P(\mathsf{H}_i)$ can be specified as the parallel composition of four sequential programs $Offer(\mathsf{H}_i)$, $Register(\mathsf{H}_i)$, $Cancel(\mathsf{H}_i)$ and $CheckOut(\mathsf{H}_i)$, for, respectively, retrieving requests and sending offers, registering confirmed reservations, cancelling declined offers and checking out the customers:

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

$$\begin{aligned}
\mathit{Offer}(\mathsf{H}_i) &\stackrel{\text{def}}{=} !\mathit{in}(\mathsf{Req}(\mathsf{N}^?, \mathsf{D}^?) : \mathtt{L})_\ \mathit{inp}(\mathsf{Avail}_i(\mathsf{D}, \mathsf{P}^?) : \mathtt{H})? \\
&\qquad \mathit{out}(\mathsf{Res}_i(\mathsf{N}, \mathsf{D}, \mathsf{P}) : \mathtt{L})_\ \mathit{out}(\mathsf{Off}(\mathsf{N}, \mathsf{H}_i, \mathsf{D}, \mathsf{P}) : \mathtt{L}).0 \\
&\qquad : 0 \\
\mathit{Register}(\mathsf{H}_i) &\stackrel{\text{def}}{=} !\mathit{in}(\mathsf{Conf}(\mathsf{H}_i, \mathsf{N}^?, \mathsf{D}^?, \mathsf{P}^?) : \mathtt{L})_\ \mathit{in}(\mathsf{Res}_i(\mathsf{N}, \mathsf{D}, \mathsf{P}) : \mathtt{L}) \\
&\qquad _\ \mathit{out}(\mathsf{Book}_i(\mathsf{N}, \mathsf{D}, \mathsf{P}) : \mathtt{H}).0 \\
\mathit{Cancel}(\mathsf{H}_i) &\stackrel{\text{def}}{=} !\mathit{in}(\mathsf{Canc}(\mathsf{H}_i, \mathsf{N}^?, \mathsf{D}^?, \mathsf{P}^?) : \mathtt{L})_\ \mathit{in}(\mathsf{Res}_i(\mathsf{N}, \mathsf{D}, \mathsf{P}) : \mathtt{L})_\ \mathit{out}(\mathsf{Avail}_i(\mathsf{D}, \mathsf{P}) : \mathtt{H}).0 \\
\mathit{Checkout}(\mathsf{H}_i) &\stackrel{\text{def}}{=} !\mathit{in}(\mathsf{Free}(\mathsf{H}_i, \mathsf{N}^?, \mathsf{D}^?, \mathsf{P}^?) : \mathtt{L})_\ \mathit{in}(\mathsf{Book}_i(\mathsf{N}, \mathsf{D}, \mathsf{P}) : \mathtt{H}) \\
&\qquad _\ \mathit{out}(\mathsf{Avail}_i(\mathsf{D}, \mathsf{P}) : \mathtt{H}).0
\end{aligned}$$

where D stands for room description and P for room price. We have assumed that each hotel H_i has two (stable) boards Avail_i and Book_i for keeping track of the rooms currently available and those already booked. Moreover, each hotel H_i has a (zero-safe) board Res_i for keeping track of the ongoing reservation agreements. The notation $\mathsf{N}^?$ in an input means that, for that field, any value is allowed in the retrieved tuple, and that such a value is then substituted for N in the continuation.

The generic process $Q(\mathsf{N}_j)$ can be specified as the parallel composition of two sequential programs $\mathit{Ask}(\mathsf{N}_j)$ and $\mathit{Choose}(\mathsf{N}_j)$, for issuing a first request for the room and accepting/declining the offers received from the hotels, respectively:

$$\begin{aligned}
\mathit{Ask}(\mathsf{N}_j) &\stackrel{\text{def}}{=} \mathit{out}(\mathsf{Req}(\mathsf{N}_j, \mathsf{D}_j) : \mathtt{L}).0 \\
\mathit{Choose}(\mathsf{N}_j) &\stackrel{\text{def}}{=} !\mathit{in}(\mathsf{Off}(\mathsf{N}_j, \mathsf{H}^?, \mathsf{D}^?, \mathsf{P}^?) : \mathtt{L})_\ \mathit{rdp}(\mathsf{Ok}_j(\mathsf{D}, \mathsf{P}) : \mathtt{H})? \\
&\qquad \mathit{out}(\mathsf{Conf}(\mathsf{H}, \mathsf{N}_j, \mathsf{D}, \mathsf{P}) : \mathtt{L}).Q'_j \\
&\qquad : \mathit{out}(\mathsf{Req}(\mathsf{N}_j, \mathsf{D}_j) : \mathtt{L})_\ \mathit{out}(\mathsf{Canc}(\mathsf{H}, \mathsf{N}_j, \mathsf{D}, \mathsf{P}) : \mathtt{L}).0
\end{aligned}$$

where we assume that each person N_j has a private board Ok_j containing the pairs (description,price) that N_j can afford. We also assume that, sooner or later, the continuation Q'_j of a successfull booking will issue the check-out message $\langle \mathsf{Free}(\mathsf{H}, \mathsf{N}_j, \mathsf{D}, \mathsf{P}) : \mathtt{L} \rangle$.

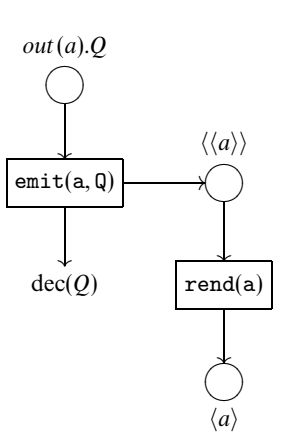
For other examples, we refer the interested reader to those given for zero-safe nets in Bruni and Montanari (2000a; 2000b) (for example, multicasting, dining philosophers), that can also be expressed naturally in TraLinda.

4.3. *Ordered semantics as transactional semantics*

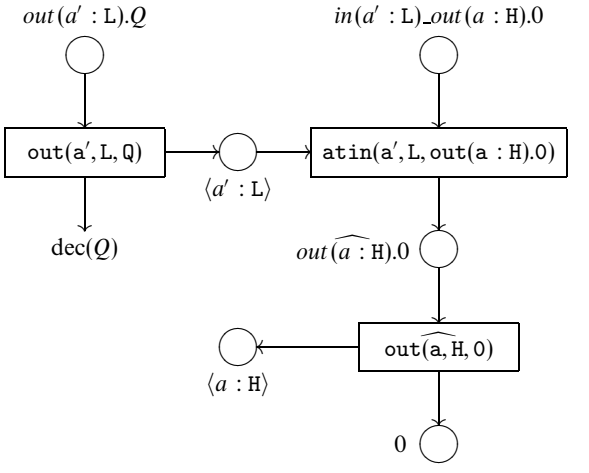
To show the role played by zero-safe nets, and, more generally, by transactions, in programming languages, let us consider the transformation div(_) from Linda programs to TraLinda programs, which sends $\mathit{out}(a).Q$ to $\mathit{out}(a' : \mathtt{L}).Q | \mathit{in}(a' : \mathtt{L})_\mathit{out}(a : \mathtt{H}).0$ (where each stable message a has a zero-safe counterpart a'), and is otherwise the identity (typing all the other messages with H).

If we abandon the distinction between stable and zero places, and consider the ordered semantics given by ri-nets $\mathit{net}(\mathrm{div}(P))$ for transformed processes $\mathrm{div}(P)$, with $a : \mathtt{H}$ and $a' : \mathtt{L}$ viewed just as different (stable) messages, and with atomic prefixing viewed as ordinary prefixing, it is easy to verify that the resulting nets are essentially the same as

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press



(a) Unordered output in P.



(b) Ordered output in (non-transactional) div(P).

Figure 19. Structural differences between P and div(P).

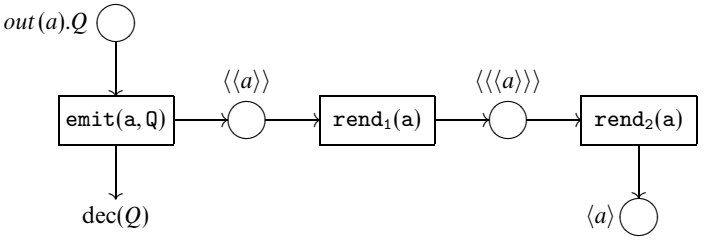


Figure 20. Delayed output in $net_D(P)$.

those associated by the unordered semantics to the original (that is, non-transformed) processes P, and therefore deadlock is decidable by results in Busi *et al.* (2000b).

Proposition 4.7. Given a Linda process P, let $net_U(P)$ denote the unordered semantics of P and $net_O(\text{div}(P))$ denote the ordered semantics of the process div(P). Then, the nets $net_O(\text{div}(P))$ and $net_U(P)$ are deadlock equivalent.

Proof. The two nets differ only in the translation of output prefixes in P. In $net_U(P)$ the output involves the transitions in Figure 19(a), while $net_O(\text{div}(P))$ models output in P as in Figure 19(b). This is the only structural difference. Note that, by construction of div(P), the place $\langle a' : \mathtt{L} \rangle$ cannot be accessed by other transitions. To prove the equivalence between the two nets, we exploit a third net, say $net_D(P)$ where the subscript stands for *delayed*, which is obtained by adding places $\langle\langle\langle a \rangle\rangle\rangle$ to $net_U(P)$ and by replacing all transitions `rend(a)` by the two transitions (see Figure 20)

$$\mathtt{rend_1(a)}\colon\ 0 \multimap \langle\langle a \rangle\rangle \xrightarrow{0} \langle\langle\langle a \rangle\rangle\rangle$$
$$\mathtt{rend_2(a)}\colon\ 0 \multimap \langle\langle\langle a \rangle\rangle\rangle \xrightarrow{0} \langle a \rangle.$$

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

In fact, $net_D(P)$ is deadlock equivalent to $net_U(P)$ (since the 'duplication' is made on places that are not connected to any inhibitor arc), while the equivalence between $net_D(P)$ and $net_O(\mathrm{div}(P))$ can be proved by establishing a bijective correspondence between reachable markings. First note that $net_O(\mathrm{div}(P))$ is, of course, equivalent to $net_O(\mathrm{div}(P|0))$ for all P. In particular, if m is the number of messages initially present in P, then we can add m tokens in the place 0 of $net_O(\mathrm{div}(P))$ without changing the behaviour of the net. Then, the bijection sends:

— $\langle a \rangle$ to $\langle a \rangle \oplus 0$;
— $\langle\langle a \rangle\rangle$ to $\langle a' : \mathtt{L} \rangle \oplus in(a' : \mathtt{L})_out(a : \mathtt{H}).0$;
— $\langle\langle\langle a \rangle\rangle\rangle$ to $out(\widehat{a : \mathtt{H}}).0$; and
— $out(a).Q$ to $out(a' : \mathtt{L}).Q \oplus in(a' : \mathtt{L})_out(a : \mathtt{H}).0$.

(In the opposite direction, note that the number of tokens in $\langle\langle a \rangle\rangle$ and in $out(a).Q$ are determined by the tokens in $\langle a' : \mathtt{L} \rangle$ and in $out(a' : \mathtt{L}).Q$, respectively.) This is easy to prove by induction as the property is preserved by firings and it holds in the initial states, where $\langle\langle a \rangle\rangle$ and $\langle\langle\langle a \rangle\rangle\rangle$ are empty and so are $\langle a' : \mathtt{L} \rangle$ and $out(\widehat{a : \mathtt{H}}).0$, while, by construction, if $n \cdot out(a).Q \in \mathrm{dec}(P)$, then $n \cdot \mathrm{dec}(out(a' : \mathtt{L}).Q | in(a' : \mathtt{L})_out(a : \mathtt{H}).0) \in \mathrm{dec}(\mathrm{div}(P))$. It follows that whenever $\mathtt{rend}_1(\mathtt{a})$ can be fired in $net_D(P)$, we have $\mathtt{atin}(\mathtt{a}', \mathtt{L}, \mathtt{out}(\mathtt{a} : \mathtt{H}).0)$ can be fired in $net_O(\mathrm{div}(P))$, and *vice versa*. Analogously, whenever $\mathtt{rend}_2(\mathtt{a})$ can be fired in $net_D(P)$, we have $\mathtt{out}(\widehat{\mathtt{a}, \mathtt{H}, 0})$ can be fired in $net_O(\mathrm{div}(P))$. Since the nets are otherwise analogous, they have the same set of computations and are therefore also deadlock equivalent. □

However, when the atomicity constraint is imposed on $\mathrm{div}(P)$ by viewing it as a TraLinda process (where the places $\langle a' : \mathtt{L} \rangle$ and $\widehat{P}$ are zero-safe), the resulting abstract net is essentially the same as the net associated to the original processes according to the ordered semantics, which defines a Turing equivalent framework.

Proposition 4.8. Given a Linda process P, let $net_O(P)$ denote the ordered semantics of P and let $A_{E(\mathrm{div}(P))}$ denote the abstract ri-net of the TraLinda process $\mathrm{div}(P)$. Then, the nets $net_O(P)$ and $A_{E(\mathrm{div}(P))}$ are deadlock equivalent.

Proof. The only zero places in the zs ri-net $E(\mathrm{div}(P))$ are those associated to messages $a' : \mathtt{L}$ and processes $out(\widehat{a : \mathtt{H}}).0$, because atomic prefixes are not introduced by the mapping $\mathrm{div}(_)$ in any other way. It follows that the only transactions in $E(\mathrm{div}(P))$ are those involving the production of a token in $\langle a' : \mathtt{L} \rangle$ and its (two-step) retrieval due to the firing of $\mathtt{atin}(\mathtt{a}', \mathtt{L}, \mathtt{out}(\mathtt{a} : \mathtt{H}).0)$ followed by $\mathtt{out}(\widehat{\mathtt{a}, \mathtt{H}, 0})$ (see Figure 21), that is, for each Q such that $out(a' : \mathtt{L}).Q$ is in $E(\mathrm{div}(P))$, we have an abstract transition from $out(a' : \mathtt{L}).Q \oplus in(a' : \mathtt{L})_out(a : \mathtt{H}).0$ to $\mathrm{dec}(Q) \oplus \langle a : \mathtt{H} \rangle \oplus 0$. On the other hand, by construction of $\mathrm{div}(P)$, whenever a token is released in $out(a' : \mathtt{L}).Q$ a token is also released in $in(a' : \mathtt{L})_out(a : \mathtt{H}).0$ and there is no other transition involving those places. Therefore, by analogy with the proof of Proposition 4.7, we can establish a correspondence between markings of $net_O(P)$ and $A_{E(\mathrm{div}(P))}$ by sending $out(a).Q$ to $out(a' : \mathtt{L}).Q \oplus in(a' : \mathtt{L})_out(a : \mathtt{H}).0$, so that the two nets are deadlock equivalent. □

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

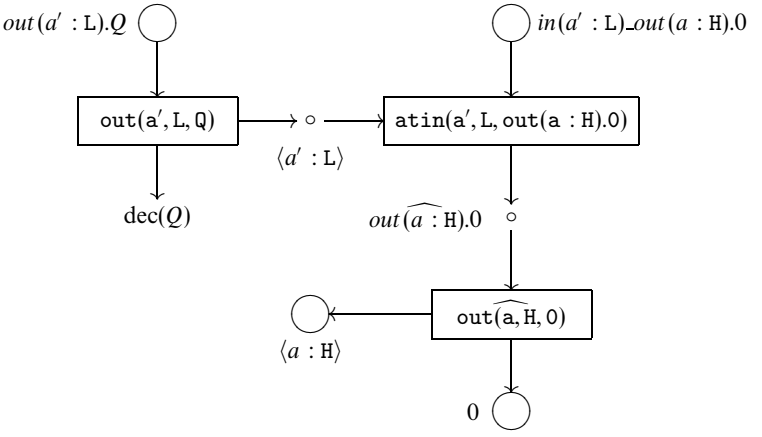


Figure 21. Transactional semantics for output in div(P).

This means that while the Linda process div(P) models the unordered version of the program P, the TraLinda process div(P) (where in the associated net all places $\langle a' : \mathtt{L}\rangle$ and $out(a : \mathtt{H}).0$ are zero-safe) is executed under the atomicity assumptions of transactions, thus recovering the ordered semantics.

4.4. *Implementability issues*

The main features of our linguistic proposal for programming transactions in a coordination language are:

(1) it comes naturally equipped with a truly concurrent semantics;
(2) it reconciles the abstract and refined views of the modelled system;
(3) new participants can dynamically join ongoing transactions, exploiting data-driven coordination.

An additional important aspect is the effectiveness of a distributed implementation for the language TraLinda, where the crux is the commit protocol for transactions. For example, BizTalk and JavaSpaces exploit centralised *transaction managers* to coordinate distributed activities, and interleaving semantics for transactional languages are often based on the assumption that the commit can be performed by an entity capable of taking control over a large (if not the whole) part of the distributed system. The point is that the presence of a transaction manager in global computing raises several issues involving both theoretical and pragmatic aspects (and possibly even ethical ones as well):

(1) locality of both data and control;
(2) the heavy task overload on the servers running transaction managers;
(3) when two or more organisations are involved in a transaction, which transaction manager should take control?

The paper Bruni *et al.* (2002a) presents a distributed implementation of zero-safe nets in the Join calculus, where a novel commit algorithm is exploited: the *distributed* 2PC. The

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

distributed 2PC is a variant of the *decentralised* 2PC protocol (Bernstein *et al.* 1987). In the decentralised 2PC, every participant to a transaction knows all the other participants involved in the same transaction. When one participant is ready to commit, it sends a message to all its companions. At the same time it collects all the commit messages coming from these companions. When all the messages have been received, the participant commits. Alternatively, if a participant wants to abort the transaction, it sends the abort message to all its companions, and rolls back to its own initial state. Every participant receiving the abort message also rolls back.

In the distributed 2PC protocol, the number and the identity of participants are not statically fixed, and participants that are ready to commit know only the participants with whom they have directly cooperated (exchanging data). As in the decentralised 2PC, the role of the coordinator is played by all participants. In particular, during the first phase of the commit protocol, every participant builds, in linear time, its own transitive closure of the processes cooperating in the transaction. When the transitive closure is stable, all participants can commit. If an abort message occurs, each participant restores its (local) initial state upon receiving the abort message.

The distributed implementation exploits all of the expressive power of the join-calculus for the dynamic definition of local transaction controllers, whose identity is extruded only to companions upon some data exchange inside the transaction. In the encoding, places become channel names $a, b, c, \ldots$, tokens become messages $a\langle x\rangle, b\langle y\rangle, c\langle z\rangle, \ldots$ and transitions become definitions $J \triangleright P$ (whose patterns J model transitions pre-sets). It is worth noting that the distributed 2PC is compiled in the encoding and thus it is completely transparent to the programmer. In fact, if a transition spawns a new participant (for example, if it opens a transaction by only consuming stable resources while producing zero-safe resources) then the body P of the corresponding join definition $J \triangleright P$ contains the declaration of local names and controller, so that whenever the rule is applied, a fresh controller will be created and released. In fact, P would have the form $\mathsf{def}\ D\ \mathsf{in}\ P'$ where D defines the fresh controller, which is local to the participant who executed $J \triangleright P$.

We are confident that the encoding in Bruni *et al.* (2002a) can be extended to ZS ri-nets by introducing additional suitable controllers for regulating the access to resources when they must be read, produced, consumed and tested for absence. Then, transitions should be encoded in such a way that resources can only be accessed through their controllers, and in the case of some violation attempt (for example, producing a resource that has been tested for absence in an ongoing transaction) the suitable counter-measures must be taken (for example, aborting the transaction).

5. One, no-one, one-hundred-thousand transactions[†]

In this section we sketch a comparison between our approach and other proposals in the literature for programming transactions over distributed networks, with the aim of shedding some light on similarities and differences. In particular, we focus on BizTalk and JavaSpaces.

[†] Thanks are due to Luigi Pirandello for inspiring the title of this section.

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

BizTalk provides users with a graphical environment called *BizTalk Orchestration Visual Designer*, which offers a flowchart view, where several basic shapes, representing the primitives of the system, can be connected together to build up applications. Transactions are defined by boxing series of shapes together. There are three kinds of transaction, called short-lived, long-running and timed. Transactions are only allowed to be nested up to two levels and not at all within short transactions. Automated recovery in case of failure is built-in for short transactions only; for long and timed transactions it is the programmer's task to define suitable compensation algorithms. A main constraint is that each transaction must have exactly one entering and one successful exit point. We have shown in joint work with Cosimo Laneve (Bruni *et al.* 2002a) that BizTalk short transactions can be easily recast in the ordinary zs net approach, without requiring read and inhibitor arcs. Moreover, zs nets manage the case where many input and output points are present (multiway transactions). A main result of Bruni *et al.* (2002a) is to define a novel commit protocol, that can be used for implementing zs nets in a completely distributed manner, as shown by, for example, the prototyping join-calculus implementation of Bruni *et al.* (2002a).

For the comparison with JavaSpaces, we rely on the process calculus presentation given by Nadia Busi and Gianluigi Zavattaro (Busi and Zavattaro 2001). Apart from a different syntax for the primitives, the calculus extends Linda with two prefixes $create(x)$ and $commit(x)$ such that:

— When $create(x).P$ is executed, a new transaction is created (uniquely identified by a fresh identifier y allocated at run time), and the residual $P[y/x]$ is wrapped inside the monitor for the transaction, which rules the access to external resources.

— When all the threads participating in the transaction y are either messages M or have the form $commit(y).P$, the commit can be performed, consuming all the prefixes $commit(y)$ and unwrapping the monitor.

The monitor must keep track of the external resources that are read and tested for absence to guarantee transaction serialisability, forbidding undesired dependencies between disjoint transactions. With respect to BizTalk, transactions still have one entry point, but can have several exit points, which must all be coordinated by the monitor for committing. Nesting is allowed in the syntax, but then forbidden by the semantics, as a fresh transaction cannot be opened inside an already running transaction. Messages produced during the transaction are only made available to the system at commit time (as the stable messages of TraLinda), but they can also be consumed by any participant into the same transaction (like the zero-safe messages of TraLinda). Moreover, each participant can read or input from the environment. When testing inside a transaction, a datum is considered absent only if it is absent both inside the wrapping and from the environment.

Figure 22 shows a tentative translation from the proposal in Busi and Zavattaro (2001) to TraLinda, which inductively maps processes P to $\mathrm{at}(P)_{\emptyset}$. The subscript X in $\mathrm{at}(P)_X$ denotes the names of the (nested) transactions occurring above P. If the program is correct, it contains no nesting and therefore X is either a singleton or the empty set. In the presence of nested transactions, the mapping just integrates lower-level transactions

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

$$
\begin{aligned}
\mathrm{at}(0)_X &\stackrel{\mathrm{def}}{=} 0\\
\mathrm{at}(\langle a\rangle)_\varnothing &\stackrel{\mathrm{def}}{=} \langle a : \mathtt{H}\rangle\\
\mathrm{at}(\langle a\rangle)_{\{x\}} &\stackrel{\mathrm{def}}{=} \langle a(x) : \mathtt{L}\rangle\\
\mathrm{at}(create(y).P)_\varnothing &\stackrel{\mathrm{def}}{=} out(\tau(x) : \mathtt{H})_\, \mathrm{at}(P[x/y])_{\{x\}} \,|\, \Big(|_{a\in \mathrm{msg}(P)_x}\, !in(a(x) : \mathtt{L})_\, out(a : \mathtt{H}).0\Big)\\
&\qquad \text{with } x \text{ fresh}\\
\mathrm{at}(create(y).P)_{\{x\}} &\stackrel{\mathrm{def}}{=} out(\tau(x) : \mathtt{H})_\, \mathrm{at}(P[x/y])_{\{x\}}\\
\mathrm{at}(in(a).P)_\varnothing &\stackrel{\mathrm{def}}{=} in(a : \mathtt{H}).\mathrm{at}(P)_\varnothing\\
\mathrm{at}(in(a).P)_{\{x\}} &\stackrel{\mathrm{def}}{=} inp(a : \mathtt{H})?\mathrm{at}(P)_{\{x\}} : in(a(x) : \mathtt{L}).\mathrm{at}(P)_{\{x\}}\\
\mathrm{at}(rd(a).P)_\varnothing &\stackrel{\mathrm{def}}{=} rd(a : \mathtt{H}).\mathrm{at}(P)_\varnothing\\
\mathrm{at}(rd(a).P)_{\{x\}} &\stackrel{\mathrm{def}}{=} rdp(a : \mathtt{H})?\mathrm{at}(P)_{\{x\}} : \underline{rd(a(x) : \mathtt{L}).\mathrm{at}(P)_{\{x\}}}\\
\mathrm{at}(out(a).P)_\varnothing &\stackrel{\mathrm{def}}{=} out(a : \mathtt{H}).\mathrm{at}(P)_\varnothing\\
\mathrm{at}(out(a).P)_{\{x\}} &\stackrel{\mathrm{def}}{=} out(a(x) : \mathtt{L}).\mathrm{at}(P)_{\{x\}}\\
\mathrm{at}(commit(x).P)_X &\stackrel{\mathrm{def}}{=} \mathrm{at}(P)_{X\setminus\{x\}}\\
\mathrm{at}(!in(a).P)_\varnothing &\stackrel{\mathrm{def}}{=} !in(a : \mathtt{H}).\mathrm{at}(P)_\varnothing\\
\mathrm{at}(rdp(a)?P : Q)_X &\stackrel{\mathrm{def}}{=} rdp(a : \mathtt{H})?\mathrm{at}(P)_X : \mathrm{at}(Q)_X\\
\mathrm{at}(inp(a)?P : Q)_X &\stackrel{\mathrm{def}}{=} inp(a : \mathtt{H})?\mathrm{at}(P)_X : \mathrm{at}(Q)_X\\
\mathrm{at}(P|Q)_X &\stackrel{\mathrm{def}}{=} \mathrm{at}(P)_X|\mathrm{at}(Q)_X
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{msg}(0)_x &\stackrel{\mathrm{def}}{=} \varnothing\\
\mathrm{msg}(\langle a\rangle)_x &\stackrel{\mathrm{def}}{=} \{a\}\\
\mathrm{msg}(out(a).P)_x &\stackrel{\mathrm{def}}{=} \{a\} \cup \mathrm{msg}(P)_x\\
\mathrm{msg}(commit(x).P)_x &\stackrel{\mathrm{def}}{=} \varnothing\\
\mathrm{msg}(\eta.P)_x &\stackrel{\mathrm{def}}{=} \mathrm{msg}(P)_x \quad \text{if } \eta \neq out(a) \wedge \eta \neq commit(x)\\
\mathrm{msg}(\mu?P : Q)_x &\stackrel{\mathrm{def}}{=} \mathrm{msg}(P)_x \cup \mathrm{msg}(Q)_x\\
\mathrm{msg}(P|Q)_x &\stackrel{\mathrm{def}}{=} \mathrm{msg}(P)_x \cup \mathrm{msg}(Q)_x
\end{aligned}
$$

Figure 22. The mapping at(_).

as part of the topmost transaction. We say 'tentative translation' because it serves as a way of illustrating some conflicting situation between the two approaches.

Computationally, the main difference is that in Busi and Zavattaro (2001) messages released inside a transaction can be either consumed by processes running in that particular transaction or (if left) made available to all other processes at commit time, while in TraLinda messages are either zero-safe, and hence apt to be consumed inside transactions, or stable, that is, available at commit time only. Therefore, the mapping at(_) introduces zero-safe messages $a(x)$ for each message a and transaction name x, where messages can be fetched by the processes involved in the transaction x but not from outsiders. On the

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

other hand, all the messages $\langle a(x)\rangle$ must have the possibility of becoming stable messages $\langle a\rangle$, otherwise the transaction cannot be closed. In the definition of $\mathrm{at}(create(y).P)_{\emptyset}$, this is accomplished by adding the process $|_{a\in \mathrm{msg}(P)_x} !in(a(x) : \mathtt{L})_\ out(a : \mathtt{H}).0$ in parallel with the translation of P, where $\mathrm{msg}(P)_x$ is in fact the set of messages that can be produced inside x by P. Note that $\mathrm{msg}(P)_x$ is not a multiset (that is, in Figure 22, the symbol $\cup$ denotes ordinary set union) and that, in general, it is a set larger than or equal to the set of messages that are produced by P in any execution of the transaction x.

Moreover, in Busi and Zavattaro (2001), when inputting or reading a inside a transaction x, processes can either select the tuple a from the environment, or select the tuple a from the tuple space that is local to x. In TraLinda we can use conditional statements to implement the choice between the two alternatives. We recall that since inhibitor arcs are not allowed on zero places, zero-safe messages are not allowed in guards; hence processes must first attempt to select a from the environment if available, otherwise they can resort to local messages $a(x)$. In the definition of $\mathrm{at}(rd(a).P)_{\{x\}}$, the process $rd(a(x) : \mathtt{L}).\mathrm{at}(P)_{\{x\}}$ is underlined because it reads from a zero-safe channel. Though this is not allowed in TraLinda, the extension of ZS r-nets with read arcs on zero places has already been considered, for example, in Bruni and Montanari (2001a), and thus the operation is consistent with the zero-safe approach (here we have chosen the simplest version for a uniform treatment of read and inhibitor arcs).

The channels $\tau(x) : \mathtt{H}$, introduced for all transaction identifiers x, are just used for producing messages recording successful transactions. They facilitate the definition of the encoding $\mathrm{at}(_\ )$. In the presence of nested transactions, several messages $\tau(x) : \mathtt{H}$ for the same identifier x can be produced. Note that being $\tau(x)$ stable, the messages will be produced together, at commit time.

A unique identifier x is assigned to each transaction. Transaction identifiers are freshly generated at 'compilation' time. This is in contrast with the run-time allocation of Busi and Zavattaro (2001). In particular, we must assume that guarded replication $!in(a).P$ cannot be used to dynamically generate fresh transactions, that is, that P does not contain $create(x)$ prefixes. This restriction is due to the fact that in ZS nets we cannot dynamically allocate places and transactions. Moreover, the process $!in(a).P$ cannot be used inside any transaction either, because the token in the place $!\widehat{in(a)}.P$ would never be consumed, and the transaction would never be able to commit.

The most important and unavoidable difference is that the conditional statement cannot be used for testing the presence of zero-safe messages. This means that $\mathrm{at}(\mu?P : Q)_X$ can only test for the presence of $\langle a : \mathtt{H}\rangle$, not for $\langle a(x) : \mathtt{L}\rangle$ (at least not without introducing inhibitor arcs on zero places, which would compromise the consistency of the abstract model). Instead, it is worth remembering that read arcs on zero places are, in principle, admissible, but we have decided not to consider them to keep the presentation of ZS ri-nets (and of TraLinda) simpler and uniform. In Section 6, where inhibitor arcs are not considered at all, we will remove this limitation, carrying the construction for the more general setting where read arcs on zero places are allowed.

From the above discussion, the approach in Busi and Zavattaro (2001) seems more general and more expressive than the zero-safe approach, at least with respect to the operations allowed inside transactions. Moreover, in Busi and Zavattaro (2001) it is also

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

shown how we can enrich the language by adding primitives for event notification. On the other hand, our approach benefits from:

(1) allowing multiway transactions instead of transactions with a unique entry point;
(2) being more distributed (as opposed to having a unique transaction coordinator for each transaction);
(3) defining the abstract view of the system;
(4) enforcing atomicity at the level of the operational semantics.

6. Algebraic semantics

Negative premises like those necessary for testing the absence of messages in the conditional statements of Linda and TraLinda are of course difficult to implement efficiently in largely distributed systems, and for this reason many distributed languages rely on primitives whose semantics is based on positive premises only (this is the case, for example, for concurrent constraint programming). In general, negative premises thus represent a trade off between efficiency and expressiveness.

In the case of TraLinda, when considering the sublanguage without conditional statements, we are able to reconcile the abstract and refined view of its net model as a categorical coreflection:

— We know that given a ZS net, the characterisation of its abstract net can be formulated as a coreflection between the category **Petri** (whose objects are PT nets and whose morphisms are graph homomorphisms sending transitions into transitions) into either the category **ZSC** (whose objects are ZS nets and whose morphisms are causal refinement morphisms sending transitions into transactions) when the individual token philosophy is considered, or into the category **ZSN** (whose objects are ZS nets and whose morphisms are refinement morphisms sending transitions into 'collective' transactions) if the collective token philosophy is selected – see Bruni and Montanari (2000b) for details.
— In this section we show how to extend the approach to ZS r-nets under the individual token philosophy. Note that ZS r-nets do not support inhibitor arcs, and thus cannot be used directly as a truly concurrent model for TraLinda. Nevertheless, they suffice for modelling the sublanguage of TraLinda consisting of processes without conditional statements. Moreover, since the main reason for avoiding read arcs on zero-safe places was to have a uniform treatment of positive/negative contexts, we can now consider the more general case where the *rd* primitive can be applied to zero-safe messages (we refer the reader to Bruni and Montanari (2001a) for the introduction of read arcs on zero-safe places).

More precisely, the (positive) transactional coordination language that we consider is defined in Figure 23, while its encoding in ZS r-nets is given by the subset of rules in Figure 16 disregarding conditional statements and the read prefix, plus the four rules in Figure 24 (it can be verified immediately that all such transitions do indeed have empty negative contexts). It is worth noting that a (non-transactional) coordination language

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

$$
\begin{array}{ll}
S ::= 0 \mid \eta.C \mid \eta_ C & \eta ::= out(a:\tau) \mid rd(a:\tau) \mid in(a:\tau) \mid !in(a:\tau) \\
C ::= S \mid C|C & \\
P ::= M \mid C \mid P|P & M ::= \langle a:\tau \rangle
\end{array}
$$

Figure 23. A transactional coordination language based on zs r-nets.

$$
\begin{aligned}
\mathtt{rd}(\mathtt{a},\tau,\mathtt{P}):\ & 0 +\!\!\circ\ rd(a:\tau).P \xrightarrow{\langle a:\tau\rangle} \mathrm{dec}(P) \\
\mathtt{rd}(\widehat{\mathtt{a}},\tau,\mathtt{P}):\ & 0 +\!\!\circ\ rd(\widehat{a:\tau}).P \xrightarrow{\langle a:\tau\rangle} \mathrm{dec}(P) \\
\mathtt{atrd}(\mathtt{a},\tau,\mathtt{P}):\ & 0 +\!\!\circ\ rd(a:\tau)_\ P \xrightarrow{\langle a:\tau\rangle} \widehat{\mathrm{dec}(P)} \\
\mathtt{atrd}(\widehat{\mathtt{a}},\tau,\mathtt{P}):\ & 0 +\!\!\circ\ rd(\widehat{a:\tau})_\ P \xrightarrow{\langle a:\tau\rangle} \widehat{\mathrm{dec}(P)}
\end{aligned}
$$

Figure 24. Basic transitions for the more general kind of read prefix.

with analogous set of primitives has been considered, for example, in Busi *et al.* (1998) (together with other possible combinations).

The main result of this section is Theorem 6.12, which is achieved by:

(i) recasting in partial membership equational logic the naive constructions for zs nets in Bruni and Montanari (2000b);
(ii) generalising the construction to deal with read arcs;
(iii) exploiting the subsorting mechanism of partial membership equational theories to characterise the meaningful computations of zs r-nets – a technique analogous to that in Bruni *et al.* (2002b).

6.1. *Petri nets, pre-nets and read arcs*

The construction of abstract nets exploits the 'Petri nets are monoids' approach initiated in Meseguer and Montanari (1990), where the computations of a net are shown to be the arrows of a strict symmetric monoidal category, with markings as objects. In fact, the functoriality of the monoidal operation with respect to sequential composition expresses the concurrency of the model, as computations that differ only in the order in which concurrent events are executed cannot be distinguished. Moreover, in the case of the collective token philosophy:

(1) The semantics is amenable to a functorial semantics, which lifts simulation morphisms between nets to monoidal functors between their categories of computations.
(2) The semantics defines an adjunction between the category of 'programs' (nets) and behaviours, thus defining initial models.

Moreover, in Bruni *et al.* (2001) it is shown that the construction can be conveniently expressed at the level of algebraic theories (Σ, E) in *partial membership equational logic* (Meseguer 1998) rather than at the level of their categories of models ((Σ, E)-algebras).

If the individual token philosophy is selected instead of the collective one, symmetries must be introduced to model all the possible causal flows of tokens in the net. Though

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

in this case the construction from PT nets is no longer functorial, it has been argued in Bruni *et al.* (2001) that *pre-nets* can be used to solve the problem. A pre-net is essentially an implementation of a PT net, where the abstract data structure of multisets is refined into a more concrete *string* structure, and where each transition $t : u \to v$ is simulated by *one* arbitrarily fixed linear implementation $t_{\bar{u},\bar{v}} : \bar{u} \to \bar{v}$, for some linearisations $\bar{u}$ and $\bar{v}$ of u and v. We remark that pre-nets have a different computational interpretation from phrase-structure grammars, since pre-nets do not distinguish between terminal and non-terminal symbols, and strings can be permuted before performing any step, that is, ordinary grammars would only generate monoidal categories, without symmetries. By resorting to pre-nets (instead of PT nets) the formal approach to the individual token philosophy benefits from several good properties:

— All the pre-net implementations of the same net share the same semantic model, that is, the semantics is independent of the choice of linearisations.
— Algebraic models of pre-nets are freely generated and therefore preserve colimit constructions on nets, yielding a compositional framework.

The theory PRENETS of *pre-nets* (where pre- and post-sets of transitions are taken in the free monoid of places instead of in the free commutative monoid) can be easily related to the theory SMONCAT of symmetric monoidal categories, accounting for the sought after extension of the 'Petri nets are monoids' approach to the individual token philosophy.

The choice of partial membership equational logic for presenting theories is motivated by the exploitation of membership predicates and subsorting for modelling objects just as a special kind of arrow and of partiality for modelling sequential composition (defined only if the codomain of the first arrow coincides with the domain of the second arrow). Moreover, the notion of tensor product of theories allows, for example, for the definition of the theory of monoidal categories as the tensor product of the theories of monoids and categories.

The extension of the approach to nets with read arcs has been considered in Bruni and Sassone (2000), by relying on non-free monoids of objects, and in Gadducci and Montanari (1998), exploiting match-share categories in place of symmetric monoidal categories. For Bruni and Sassone (2000), which deals satisfactorily with both the collective and the individual token philosophy, one concern is that taking non-free monoids of objects could be unrealistic, as tokens are traditionally seen as atomic pieces of data that one should not be able to decompose. Moreover, when viewing places as types, and tokens as instances, the algebraic structure imposed on places increases the complexity of the underlying type theory. For the construction in Gadducci and Montanari (1998), the problem is instead that the freely generated model of computations has too many arrows, representing computations that the net will never be able to perform. We shall sketch the construction under the individual token philosophy for pre-nets with read arcs.

The idea in Gadducci and Montanari (1998) is to model multiple concurrent readings by introducing suitable (non-natural) transformations that take care of the creation of as many copies as needed (*sharing* phase) and then the reassembly of all copies after the reading (*matching* phase). These two transformations are denoted by ∇ and Δ, respectively. Thus a transition $t : u \xrightarrow{w} v$ is first regarded as an ordinary pre-net transition

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

```
*** domain and codomain
ops d(_) c(_): Arrow -> Object.
*** monoidal product and unit
op  _⊗_ : Arrow -> Object.
op  e : Object.
*** arrow composition (partial operator)
op  _;_ .
*** symmetric natural transformation
op  γ(_,_): Object Object -> Arrow.
*** duplicator (not natural)
op  ∇(_): Object -> Arrow.
*** coduplicator (not natural)
op  Δ(_): Object -> Arrow.
```

Figure 25. Operators in MSCAT.

$[t]: u \otimes w \longrightarrow v \otimes w$, and then the free construction can be applied to the resulting pre-net, building a match-share category of computations. Note that while the product $\oplus$ is commutative (it stands for multiset union), the product $\otimes$ is not (it stands for string concatenation), as we are working with pre-nets.

For convenience and to keep this paper self-contained, the essentials of partial membership logic together with the theories of categories, monoidal categories, symmetric monoidal categories and match-share categories are summarised in Appendices A and B, respectively. Here we just remark that the theory of symmetric monoidal categories SMONCAT includes two sorts Object and Arrow (with Object a subsort of Arrow, written Object $<$ Arrow), and six operators satisfying the axioms of symmetric monoidal categories, while the theory of match-share categories MSCAT is a conservative extension of SMONCAT, with two additional operators called duplicators and coduplicators, denoted by ∇ and Δ, respectively (see Figure 25), that satisfy the axioms of match-share categories.

As notation, given a pre-net with read arcs R, we let $[R]$ denote the pre-net with the same places as R and transitions $\{[t]: u \otimes w \longrightarrow v \otimes w \mid t: u \xrightarrow{w} v \in R\}$. Unfortunately, the match-share category freely generated from $[R]$ possesses too many computations, since the model is no longer resource-conscious: the distinction between read arcs and pre/post-sets is lost and each token can be matched and shared in all possible ways. On the other hand, we also have the free symmetric monoidal category of computations for the same pre-net $[R]$, which instead distinguishes too much, in the sense that it serialises all multiple readings of the same resource. However, there is a mapping from one category to the other that solves the problem by:

(1) identifying all computations that are distinguished because of the order in which multiple readings are performed;
(2) selecting only the admissible computations of the r-net.

Definition 6.1. Let $\mathbf{S}([R])$ be the free symmetric monoidal category generated by $[R]$ and let $\mathbf{MS}([R])$ be the free match-share category generated by $[R]$. The symmetric monoidal

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

```
fth RCOMP is including MSCAT.
 sorts Rtrans Rarrow.
 subsorts Object < Rtrans < Rarrow < Arrow.
 ops pre(_) post(_) ctx(_) : Rtrans -> Object.
 op  mk(_) : Rtrans -> Arrow.
 vars h, k : Rarrow.
 var     t : Rtrans.
 var     u : Object.
 mb  h⊗k : Rarrow.
 mb  γ(u,v) : Rarrow.
 cmb h;k : Rarrow if c(h) = d(k).
 eq  pre(t)⊗ctx(t) = d(t).
 eq  post(t)⊗ctx(t) = c(t).
 eq  pre(u) = e.
 eq  post(u) = e.
 eq  ctx(u) = u.
 eq  d(mk(t)) = d(t).
 eq  c(mk(t)) = c(t).
 eq  mk(u) = u.
 eq  (pre(t)⊗∇(ctx(t)));(mk(t)⊗ctx(t));(post(t)⊗Δ(ctx(t))) = t.
endfth.
```

Figure 26. The theory RCOMP.

functor $\mathscr{E}: \mathbf{S}([R]) \to \mathbf{MS}([R])$ is defined on generators as

$$\mathscr{E}(a) \stackrel{\text{def}}{=} a$$
$$\mathscr{E}([t]) \stackrel{\text{def}}{=} (u \otimes \nabla_w); ([t] \otimes w); (v \otimes \Delta_w)$$

for any place a and transition $t: u \stackrel{w}{\longrightarrow} v \in R$.

The subcategory $\mathscr{E}(\mathbf{S}([R])) \subseteq \mathbf{MS}([R])$ is isomorphic (*via* a symmetric monoidal functor) to the category of concatenable contextual processes of R. The problem is then how to verify whether an arrow of $\mathbf{MS}([R])$ belongs to $\mathscr{E}(\mathbf{S}([R]))$.

In joint work with Meseguer and Sassone (Bruni *et al.* 2002b) we improved on Gadducci and Montanari (1998), by selecting a suitable theory of computations in which membership predicates provide the typing discipline for distinguishing between meaningful and meaningless computations among the arrows of the free match-share category $\mathbf{MS}([R])$.

The proposal in Bruni *et al.* (2002b) is to exploit a more complex theory in partial membership equational logic in order to express the above-mentioned typing discipline. Skipping the details, the theory MSCAT (see Figure 35 in Appendix B) is refined into the theory RCOMP (Figure 26) by adding two sorts Rtrans and Rarrow, with Object < Rtrans < Rarrow < Arrow: sort Rtrans is for embedding basic transitions and sort Rarrow is for collecting all correct computations. Roughly, when the operator mk(_) is applied to a transition t : Rtrans, it returns an element of sort Arrow that plays the role of

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

```
fth MON is
 sort Monoid.
 op e : -> Monoid.
 op _⊗_ : Monoid Monoid -> Monoid [assoc id: e].
endfth

fth RAUT is including MON.
 sort Rtrans.
 subsort Monoid < Rtrans.
 ops pre(_) post(_) ctx(_) : Rtrans -> Monoid.
 var u : Monoid.
 eq  pre(u) = e.
 eq  post(u) = e.
 eq  ctx(u) = u.
endfth.

view R from RAUT to RCOMP is
 sort Monoid to Object.
endview.
```

Figure 27. The theory RAUT and the theory morphism R.

[t]. In particular, the last equation states how t and mk(t) are related by matching-sharing the context of t. The free construction is then accomplished by the theory morphism R in Figure 27 from the theory RAUT of 'programs' (that is, of pre-nets with read arcs) to the theory RCOMP.

Theorem 6.2 (*cf.* Bruni *et al.* (2002b)). Given an r-net R, let $\mathscr{U}_{\mathtt{R}}$ be the forgetful functor from the category of RCOMP-algebras to the category of RAUT-algebras induced by the theory morphism R, and let $\mathscr{F}_{\mathtt{R}}$ be the left adjoint of $\mathscr{U}_{\mathtt{R}}$. Let Rarrow(R) denote the lluf subcategory of $\mathscr{F}_{\mathtt{R}}(R)$ whose arrows have sort Rarrow. Then $\mathscr{E}(\mathbf{S}([R]))$ is isomorphic (*via* a symmetric monoidal functor) to Rarrow(R).

We recall that a *lluf* subcategory **A** of a category **C** is just a subcategory having all the objects of **C** (the terminology comes from the mirrored term *full* that denotes a subcategory **A** having a subset of the objects of **C** but all the arrows between them in **C**).

Theorem 6.2 provides r-nets with a typing discipline for selecting the admissible computations from the larger class **MS**([R]). Since membership predicates allow for automated verification in languages like Maude, the construction R answers to the ambiguity of $\mathscr{E}$.

Then, having the free construction at hand, we can apply it to zs r-nets to characterise their abstract nets *via* a coreflection. Note that this construction also works for nets with reading on zero places, and it is, in fact, designed for context concatenation.

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

6.2. *Abstract semantics of* ZS *rp-nets as a coreflection*

In this section we exploit the representation results of Theorem 6.2 to give an algebraic presentation of abstract models for ZS r-nets under the individual token philosophy. Again, we work with states that are strings instead of multisets, as the symmetries introduce all the possible rearrangements.

Definition 6.3. A ZS rp-net D is a ZS r-net where for any transition t the projections on zero places of the preset, of the postset and of the context of t are strings instead of (multi)sets.

The above definition is equivalent to assuming pre-, post- and read sets of transitions to be pairs (u, x) with $u \in L^{\oplus}$ and $x \in Z^{\otimes}$ for L the set of stable places and Z the set of zero places (where $\oplus$ is commutative, and $\otimes$ is not). Note that D is a pre-net with respect to zero places only, and not for stable places.

The first step is the characterisation of transactions. In Bruni and Montanari (2000b) this has been done for ZS nets (without read arcs) by defining a concatenation operation that behaves as sequential composition on zero places and as parallel composition on stable places, like the one in Figure 6 (where, additionally, the stable context is shared between the two steps).

In the case of ZS rp-nets, the presence of read arcs requires the possibility of merging contexts. All this can be summed up by extending the theories `RAUT` and `RCOMP` with information about the stable places exploited in the computations, as illustrated in Figures 28 and 29. Note that we are interested in computations where the token flows through zero places only, for example, we want neither to concatenate computations on stable places nor to have symmetries and (co)duplicators associated to stable places.

For stable pre- and postsets, we define the theory `STABLE` (which is a convenient renaming for the theory `CMON`, whose models are commutative monoids), whilst for stable read arcs, we take `SET`. In fact, for stable places we can rely on the maximal sharing hypothesis of Bruni and Sassone (2000), that is, two concurrent transitions accessing a stable place in reading can be assumed to be reading the same token (without loss of generality for the constructions we consider).

The theory `ZSRAUT` extends `RAUT` with three operators `spre(_)`, `spost(_)` and `sctx(_)` that represent the stable preset of a transition, the stable postset and the read context, respectively. Note that by including the specification `RAUT`, the theory `ZSRAUT` possesses all the sorts and operators of `RAUT`. For example, the subsorting `Monoid` < `Rtrans` of `RAUT` means that all elements of sort `Zero` in `ZSRAUT` can be regarded as elements of sort `Rtrans`, because `Monoid` is renamed `Zero` when `RAUT` is included in `ZSRAUT`. Thus, the operators `spre(_)`, `spost(_)` and `sctx(_)` are also defined on zero places, and three axioms are added to return nil values when they are applied to elements of sort `Zero`.

Likewise, the theory `ZSRCOMP` extends `RCOMP` with three operators `spre(_)`, `spost(_)` and `sctx(_)` defined over `Rarrow` (not just over `Rtrans`). The renaming of sequential composition by `*` emphasises the similarity to the *horizontal composition* of Bruni and Montanari (2000b), which composes sequentially on zero places, but in parallel on stable

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

```
fth CMON is
 sort Cmonoid.
 op 0 : -> Cmonoid.
 op _⊕_ : Cmonoid Cmonoid -> Cmonoid [assoc comm id: 0].
endfth

fth SET is
 sort Set.
 op ∅ : -> Set.
 op _∪_ : Set Set -> Set [assoc comm idem id: ∅].
endfth

fth STABLE is CMON (renamed by sort Cmonoid to Stable.).
endfth

fth ZSRAUT is including RAUT (renamed by sort Monoid to Zero.).
              including STABLE.
              including SET.
 ops spre(_) spost(_) : Rtrans -> Stable.
 op  sctx(_) : Rtrans -> Set.
 var u : Zero.
 eq  spre(u) = 0.
 eq  spost(u) = 0.
 eq  sctx(u) = ∅.
endfth.
```

Figure 28. The theory ZSRAUT.

places. Note that the multiplicities of tokens read from stable places are abstracted away by the operator $\cup$, which is idempotent.

Proposition 6.4. The view ZR in Figure 30 defines a theory morphism.

Given a ZS rp-net D, we let Rarrow(D) be the set of elements of sort Rarrow in the ZSRCOMP-algebra $\mathscr{F}_{\mathtt{ZR}}(D)$, where $\mathscr{F}_{\mathtt{ZR}}$ is the left adjoint of the forgetful functor induced by the theory morphism ZR in Figure 30. An element f of Arrow(D) is called *prime* if $f \neq \mathtt{e}$ and whenever $f = f_1 \otimes f_2$, either $f_1 = \mathtt{e}$ or $f_2 = \mathtt{e}$. The fact that an element is prime guarantees that it cannot be decomposed into smaller separate activities.

By the result in Theorem 6.2, we have the following corollary.

Corollary 6.5. The set of (individual token) transactions of a ZS rp-net D is isomorphic to the set of prime arrows h in Rarrow(D) such that $\mathtt{d}(h) = \mathtt{c}(h) = \mathtt{e}$.

In fact, by taking prime arrows with $\mathtt{d}(h) = \mathtt{c}(h) = \mathtt{e}$ we are selecting atomic activities that start from a marking with no zero tokens and end in a marking with no zero tokens.

Note that the fact that two transitions access the same stable resource in reading does not mean that they are participating in the same transaction.

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

```
fth ZSRCOMP is including RCOMP (renamed by op ; to *.).
               including STABLE.
               including SET.
 ops spre(_) spost(_) : Rarrow -> Stable.
 op  sctx(_) : Rarrow -> Set.
 vars h, k : Rarrow.
 vars a, b : Object.
 eq  spre(h*k) = spre(h)⊕spre(k).
 eq  spre(h⊗k) = spre(h)⊕spre(k).
 eq  spre(γ(a,b)) = 0.
 eq  spre(∇(a)) = 0.
 eq  spre(Δ(a)) = 0.

 eq  spost(h*k) = spost(h)⊕spost(k).
 eq  spost(h⊗k) = spost(h)⊕spost(k).
 eq  spost(γ(a,b)) = 0.
 eq  spost(∇(a)) = 0.
 eq  spost(Δ(a)) = 0.

 eq  sctx(h*k) = sctx(h)∪sctx(k).
 eq  sctx(h⊗k) = sctx(h)∪sctx(k).
 eq  sctx(γ(a,b)) = ∅.
 eq  sctx(∇(a)) = ∅.
 eq  sctx(Δ(a)) = ∅.
endfth.
```

Figure 29. The theory ZSRCOMP.

```
view ZR from ZSRAUT to ZSRCOMP is
 sort Zero to Object.
endview.
```

Figure 30. The theory morphism ZR.

An important property noticed in Bruni *et al.* (2002b) is that for any arrow $h \in \mathtt{Rarrow}(D)$ it is possible to distinguish which resources in $\mathtt{d}(h)$ are just read (and therefore are also present in $\mathtt{c}(h)$). In fact, suppose that $\mathtt{d}(h) = u \otimes w$ and $\mathtt{c}(h) = v \otimes w$. If w has been just read, then, from the properties of the match-share categories, it must be the case that $(u \otimes \nabla(w)) * (h \otimes w) * (v \otimes \Delta(w)) = h$, that is, adding another idle copy of the context results in the same process h. In general, the position of w can be shifted in $\mathtt{d}(h)$ and $\mathtt{c}(h)$, hence we have the following characterisation.

Definition 6.6. Let $h \in \mathtt{Rarrow}(D)$ and let a be an object with $\mathtt{d}(h) = u_1 \otimes a \otimes u_2$ and $\mathtt{c}(h) = v_1 \otimes a \otimes v_2$ for suitable objects u_1, u_2, v_1, v_2. Then, a is said to be read in h if

$$(u_1 \otimes u_2 \otimes \nabla(a)) * (u_1 \otimes \gamma(u_2, a)) * (h \otimes a) * (v_1 \otimes \gamma(a, v_2)) * (v_1 \otimes v_2 \otimes \Delta(a)) = h.$$

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

The arrow $h \in \mathtt{Rarrow}(D)$ is *pure* if $\mathtt{d}(h) = u \otimes w$ and $\mathtt{c}(h) = v \otimes w$, with $(u \otimes \nabla(w)) * (h \otimes w) * (v \otimes \Delta(w)) = h$ and no other object in u and v is read.

The object w is called the *context* of h and denoted by $\mathtt{ctx}(h)$, while u and v are denoted $\mathtt{pre}(h)$ and $\mathtt{post}(h)$, respectively.

Definition 6.7. The abstract net A_D of a zs rp-net D is the r-net defined as follows:

— its places are the stable places of D;
— its transitions are the prime elements h of $\mathtt{Rarrow}(D)$ such that $\mathtt{d}(h) = \mathtt{c}(h) = \mathtt{e}$;
— for any transition h, its pre-set is $\mathtt{spre}(h)$;
— for any transition h, its post-set is $\mathtt{spost}(h)$;
— for any transition h, its read arcs are given by $\mathtt{sctx}(h)$.

The last result presented here is the characterisation of the abstract net in terms of a coreflection from the category of r-nets into the category of zs rp-nets with refinement morphisms. We start by recalling the definition of the category of r-nets and ordinary morphisms between them.

Definition 6.8. Given two r-nets R_1 and R_2, an *r-net morphism* is a pair $F = (F_{\mathrm{pl}}, F_{\mathrm{tr}})$, where $F_{\mathrm{pl}} : S_1^{\oplus} \to S_2^{\oplus}$ is a monoid homomorphism (sending places to markings) and $F_{\mathrm{tr}} : T_1 \to T_2$ is a function satisfying (for any transition $t \in T_1$):

— $\mathrm{pre}(F_{\mathrm{tr}}(t)) = F_{\mathrm{pl}}(\mathrm{pre}(t))$;
— $\mathrm{post}(F_{\mathrm{tr}}(t)) = F_{\mathrm{pl}}(\mathrm{post}(t))$;
— $\mathrm{ctx}(F_{\mathrm{tr}}(t)) = \lfloor F_{\mathrm{pl}}(\mathrm{ctx}(t)) \rfloor$.

The category **RPetri** has r-nets as object and r-net morphisms as arrows.

Next, we define the category of zs rp-nets with refinement morphisms between them. We recall the observation after Definition 6.3, which allows us to view $\mathrm{pre}(t)$, $\mathrm{post}(t)$ and $\mathrm{ctx}(t)$ as pairs in $L^{\oplus} \times Z^{\otimes}$.

Definition 6.9. Given two zs rp-nets D_1 and D_2, a *refinement morphism* $F : D_1 \to D_2$ is a triple $F = (F_{\mathrm{st}}, F_{\mathrm{z}}, F_{\mathrm{ref}})$, where $F_{\mathrm{st}} : L_1^{\oplus} \to L_2^{\oplus}$ is a monoid homomorphism (sending stable places to stable markings), $F_{\mathrm{z}} : Z_1^{\otimes} \to Z_2^{\otimes}$ is a monoid homomorphism (sending zero places to strings of zero places), and $F_{\mathrm{ref}} : T_1 \to \mathtt{Rarrow}(D)$ is a function sending transitions to pure prime arrows and satisfying (for any transition $t \in T_1$):

— $(\mathtt{spre}(F_{\mathrm{ref}}(t)), \mathtt{pre}(F_{\mathrm{ref}}(t))) = (F_{\mathrm{st}}, F_{\mathrm{z}})(\mathrm{pre}(t))$;
— $(\mathtt{spost}(F_{\mathrm{ref}}(t)), \mathtt{post}(F_{\mathrm{ref}}(t))) = (F_{\mathrm{st}}, F_{\mathrm{z}})(\mathrm{post}(t))$;
— $(\lfloor \mathtt{sctx}(F_{\mathrm{ref}}(t)) \rfloor, \mathtt{ctx}(F_{\mathrm{ref}}(t))) = \lfloor (F_{\mathrm{st}}, F_{\mathrm{z}})(\mathrm{ctx}(t)) \rfloor$.

Note that by a result in Bruni *et al.* (2002b), any transition is pure, and, of course, it is also prime.

Proposition 6.10. Refinement morphisms can be composed. Their composition is associative and has trivial identities.

Definition 6.11. The category **ZSR** has zs rp-nets as object and refinement morphisms as arrows.

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

Theorem 6.12. The category **RPetri** is embedded in **ZSR** as a coreflective subcategory. Moreover, the right adjoint functor $\mathscr{A}: \mathbf{ZSR} \to \mathbf{RPetri}$ maps each ZS rp-net D to (an r-net isomorphic to) its abstract r-net A_D.

Proof. The proof is long but analogous to that of Theorem 3.4 in Bruni and Montanari (2000b) regarding the abstract semantics of ZS nets (without read arcs) according to the collective token philosophy. In particular, several lemmas are needed stating, for example, that refinement morphisms preserve pure and prime arrows and that the abstract net of an r-net is (isomorphic to) the r-net itself. Then we show that **RPetri** is a full subcategory of **ZSR**. Finally, we prove that the obvious inclusion functor has a right adjoint $\mathscr{A}$, and that $\mathscr{A}$ sends each ZS rp-net to its abstract counterpart. This amounts to proving that for any ZS rp-net D, any r-net R and any refinement morphism $F: R \to D$, a unique morphism $G: R \to A_D$ exists such that $G; \epsilon_D = F$, where ϵ_D is the counit component of the adjunction, sending each transition of A_D into the corresponding transaction (that is, a pure and prime arrow h with $\mathsf{d}(h) = \mathsf{c}(h) = e$) of D. But this follows by Corollary 6.5 and Definition 6.7. □

We conclude by pointing out that the extension of the algebraic approach to nets with inhibitor arcs would turn the tensor product of computations into a partial operation (see, for example, Desel *et al.* (2001)), breaking up both the approach of this paper and that of Bruni and Sassone (2001).

Given that the unordered semantics is more realistic than the ordered one when largely distributed systems are considered, it is an interesting open problem to see whether inhibitor arcs really increase the expressive power of the framework, or if, instead, it is possible to find alternative but *equivalent* semantics for (unordered) TraLinda based just on ZS r-nets (that is, without inhibitor arcs), where 'equivalent' means that certain interesting features (for example, deadlocks) are exactly preserved (thus recasting the results of Busi *et al.* (2000b) and Busi (1998) in our framework).

7. Conclusion

We have shown how to handle inhibitor arcs inside the zero-safe approach, with application to the modelling of transactions in the original extension TraLinda of the language Linda.

The results presented here are part of a broader research programme, which aims to integrate the zero-safe approach with distributed languages such as the Join calculus or the JavaSpaces middleware to obtain a general purpose environment where distributed transactions can be faithfully designed, programmed and executed. In particular, we have compared our approach with the primitives considered in Busi and Zavattaro (2001) for passing from Linda-like coordination languages to JavaSpaces. Moreover, though *ad hoc* transaction mechanisms are integrated in languages such as BizTalk and JavaSpaces, we think that the zero-safe approach can provide a uniform method for adding transactions to the many calculi proposed in the literature, exploiting, for example, other enrichments of the basic net models. For example, *coloured* or *high-level* nets allow for the use of 'structured data' as tokens and their mixing with zero places presents no particular difficulty; in *reconfigurable* nets, transition postsets may depend on the values collected by the presets, addressing network reconfigurability; in *dynamic* nets, not only can a firing

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

modify the current marking, but it can also increase the set of transitions (that is, the control). The different flavours of nets are too many to be mentioned here (for example, with time, probability, priorities). An interesting correspondence between some of the above models and typed subclasses of Join agents (Fournet *at al.* 1996) has been hinted at in Asperti and Busi (1996) and formally drawn in Buscemi and Sassone (2000). The extension of the zero-safe approach to the Join calculus is hinted at in Bruni *et al.* (2002a) and deserves further study.

Regarding the implementability of TraLinda, we think that the distributed commit protocol proposed in Bruni *et al.* (2002a) can be naturally extended to TraLinda: essentially it exploits a controller for each participant in the transaction, which in the case of TraLinda can be either a transactional (sequential) process or a zero-safe tuple $\langle a : \mathtt{L} \rangle$. The presence of inhibitor arcs would probably correspond to the insertion of other controllers, one for each stable tuple $\langle a : \mathtt{H} \rangle$, in order to guarantee the consistency of the state.

Appendix A. Essentials of partial membership equational logic

In this appendix we recall some basic notions of *partial membership equational logic* (**PMEqtl**), focusing on the *one-kinded* case. A detailed exposition for the many-kinded case can be found in Meseguer (1998).

Definition A.1. A *signature* is a triple $\Omega = (\mathtt{S}, \leqslant, \Sigma)$, with $(\mathtt{S}, \leqslant)$ a poset with a top element $\top$, and $\Sigma = \{\Sigma_k\}_{k \in \mathbb{N}}$ a family of sets indexed by natural numbers. The poset $(\mathtt{S}, \leqslant)$ is called the poset of *sorts* of Ω.

Definition A.2. Given a signature $\Omega = (\mathtt{S}, \leqslant, \Sigma)$, a *partial* Ω*-algebra* A assigns:

1 to each $\mathtt{s} \in \mathtt{S}$ a set $A_{\mathtt{s}}$, in such a way that whenever $\mathtt{s} \leqslant \mathtt{s}'$, we have $A_{\mathtt{s}} \subseteq A_{\mathtt{s}'}$;
2 to each $f \in \Sigma_k$, $k \geqslant 0$, a partial function $A_f : A_\top^k \rightharpoonup A_\top$.

An Ω*-homomorphism* between two partial Ω-algebras A and B is a function $h : A_\top \to B_\top$ such that:

1 for each $\mathtt{s} \in \mathtt{S}$, $h(A_{\mathtt{s}}) \subseteq B_{\mathtt{s}}$;
2 for each $f \in \Sigma_k$, $k \geqslant 0$, and $\vec{a} \in A_\top^k$, if $A_f(\vec{a})$ is defined, then $B_f(h^k(\vec{a}))$ is also defined and equal to $h(A_f(\vec{a}))$.

This determines a category $\mathbf{PAlg}_\Omega$ of partial Ω-algebras and their morphisms.

Definition A.3. Let $\Omega = (\mathtt{S}, \leqslant, \Sigma)$ be a signature. Given a set of variables $X = \{x_1, \ldots, x_m\}$, a *variable declaration* $\tilde{X}$ is a sequence $x_1 : \mathtt{S}_1, \ldots, x_m : \mathtt{S}_m$, where for each $i \in [1, m]$, we have $\mathtt{S}_i$ is a set of sorts $\{\mathtt{s}_{i1}, \ldots, \mathtt{s}_{il_i}\}$.

Atomic Ω*-formulas* are either equations $\mathtt{t} = \mathtt{t}'$, where $\mathtt{t}, \mathtt{t}' \in \mathbb{T}_\Sigma(X)$ (with $\mathbb{T}_\Sigma(X)$ the usual free Σ-algebra on variables X) or membership assertions of the form $\mathtt{t} : \mathtt{s}$, where $\mathtt{t} \in \mathbb{T}_\Sigma(X)$ and $\mathtt{s} \in \mathtt{S}$.

General Ω*-sentences* are then Horn clauses of one of the two forms

$$(\forall \tilde{X}) \quad \mathtt{t} = \mathtt{t}' \iff \mathtt{t}_1 = \mathtt{t}'_1 \wedge \cdots \wedge \mathtt{t}_n = \mathtt{t}'_n \wedge \mathtt{t}''_1 : \mathtt{s}_1 \wedge \cdots \wedge \mathtt{t}''_m : \mathtt{s}_m$$

$$(\forall \tilde{X}) \quad \mathtt{t} : \mathtt{s} \iff \mathtt{t}_1 = \mathtt{t}'_1 \wedge \cdots \wedge \mathtt{t}_n = \mathtt{t}'_n \wedge \mathtt{t}''_1 : \mathtt{s}_1 \wedge \cdots \wedge \mathtt{t}''_m : \mathtt{s}_m$$

where the $\mathtt{t}$, $\mathtt{t}'$, $\mathtt{t}_i$, $\mathtt{t}'_i$ and $\mathtt{t}''_j$ are terms in $\mathbb{T}_\Sigma(X)$ and the $\mathtt{s}_j$ are sorts in $\mathtt{S}$.

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

Given a partial Ω-algebra A and a variable declaration $\tilde{X}$, we can define assignments $\sigma : \tilde{X} \to A$ in the obvious way (if $x_i : \mathtt{S}_i$, and $\mathtt{s} \in \mathtt{S}_i$, we must have $\sigma(x) \in A_{\mathtt{s}}$) and then we can define a partial function $\overline{\sigma} : \mathbb{T}_\Sigma(X) \rightharpoonup A_\top$, extending σ in a unique way. For atomic sentences, we then define satisfaction by:

— $A, \sigma \models \mathtt{t} = \mathtt{t}'$, meaning that $\overline{\sigma}(\mathtt{t})$ and $\overline{\sigma}(\mathtt{t}')$ are both defined and $\overline{\sigma}(\mathtt{t}) = \overline{\sigma}(\mathtt{t})$, and
— $A, \sigma \models \mathtt{t} : \mathtt{s}$ meaning that $\overline{\sigma}(\mathtt{t})$ is defined and $\overline{\sigma}(\mathtt{t}) \in A_{\mathtt{s}}$.

Satisfaction of Horn clauses is then defined in the obvious way.

Definition A.4. Given a set Γ of Ω-sentences, we let $\mathbf{PAlg}_{\Omega,\Gamma}$ be the full subcategory of $\mathbf{PAlg}_\Omega$ determined by those partial Ω-algebras that satisfy all the sentences in Γ. (The pair $\mathtt{T} = (\Omega, \Gamma)$ is a *theory*, and $\mathbf{PAlg}_\mathtt{T} = \mathbf{PAlg}_{\Omega,\Gamma}$ is the category of its models.)

Definition A.5. Given two signatures $\Omega = (\mathtt{S}, \leqslant, \Sigma)$ and $\Omega' = (\mathtt{S}', \leqslant', \Sigma')$, a *signature morphism* $\mathtt{H} : \Omega \to \Omega'$ is given by:

1 a monotonic function $\mathtt{H} : (\mathtt{S}, \leqslant) \to (\mathtt{S}', \leqslant')$, and
2 a family of functions $\{\mathtt{H}_k : \Sigma_k \to \Sigma'_k\}_{k \in \mathbb{N}}$.

A signature morphism $\mathtt{H}$ induces a forgetful functor $\mathscr{U}_\mathtt{H} : \mathbf{PAlg}_{\Omega'} \to \mathbf{PAlg}_\Omega$, where for each $A' \in \mathbf{PAlg}_{\Omega'}$ we have:

1 for each $\mathtt{s} \in \mathtt{S}$, $\mathscr{U}_\mathtt{H}(A')_\mathtt{s} = A'_{\mathtt{H}(\mathtt{s})}$;
2 for each $f \in \Sigma_k$, $\mathscr{U}_\mathtt{H}(A')_f = A'_{\mathtt{H}(f)} \cap (\underbrace{A'_{\mathtt{H}(\top)} \times \ldots \times A'_{\mathtt{H}(\top)}}_{k} \times A'_{\mathtt{H}(\top)})$;
3 for each Ω'-homomorphism $h' : A' \to B'$, $\mathscr{U}_\mathtt{H}(h') = h'|_{\mathtt{H}(\top)} : A'_{\mathtt{H}(\top)} \to B'_{\mathtt{H}(\top)}$, which is well-defined as a restriction of h' because h' is sort-preserving.

Definition A.6. Given two theories $\mathtt{T} = (\Omega, \Gamma)$ and $\mathtt{T}' = (\Omega', \Gamma')$, a *theory morphism* $\mathtt{H} : \mathtt{T} \to \mathtt{T}'$ is a signature morphism $\mathtt{H} : \Omega \to \Omega'$ such that $\mathscr{U}_\mathtt{H}(\mathbf{PAlg}_{\mathtt{T}'}) \subseteq \mathbf{PAlg}_\mathtt{T}$, so $\mathscr{U}_\mathtt{H}$ restricts to a forgetful functor $\mathscr{U}_\mathtt{H} : \mathbf{PAlg}_{\mathtt{T}'} \to \mathbf{PAlg}_\mathtt{T}$.

Proposition A.7. The forgetful functor $\mathscr{U}_\mathtt{H} : \mathbf{PAlg}_{\mathtt{T}'} \to \mathbf{PAlg}_\mathtt{T}$ associated to a theory morphism $\mathtt{H} : \mathtt{T} \to \mathtt{T}'$ has a left adjoint $\mathscr{F}_\mathtt{H} : \mathbf{PAlg}_\mathtt{T} \to \mathbf{PAlg}_{\mathtt{T}'}$.

Analogously to Lawvere algebraic theories, given two theories $\mathtt{T}$ and $\mathtt{T}'$ in partial membership equational logic there is a theory $\mathtt{T} \otimes \mathtt{T}'$ in partial membership equational logic such that $\mathbf{PAlg}_{\mathtt{T} \otimes \mathtt{T}'} \simeq \mathbf{PAlg}_\mathtt{T}(\mathbf{PAlg}_{\mathtt{T}'}) \simeq \mathbf{PAlg}_{\mathtt{T}'}(\mathbf{PAlg}_\mathtt{T})$. The explicit definition of $\mathtt{T} \otimes \mathtt{T}'$ is as follows.

Definition A.8. Let $\mathtt{T} = (\Omega, \Gamma)$ and $\mathtt{T}' = (\Omega', \Gamma')$ be theories in **PMEqtl** with $\Omega = (\mathtt{S}, \leqslant, \Sigma)$ and $\Omega' = (\mathtt{S}', \leqslant', \Sigma')$. Then their *tensor product* $\mathtt{T} \otimes \mathtt{T}'$ is the theory with signature $\Omega \otimes \Omega'$ having:

1 poset of sorts $(\mathtt{S}, \leqslant) \times (\mathtt{S}', \leqslant')$;
2 signature $\Sigma \otimes \Sigma'$, with an operator $f^l \in (\Sigma \otimes \Sigma')_n$ for each $f \in \Sigma_n$, and with an operator $g^r \in (\Sigma \otimes \Sigma')_m$ for each $g \in \Sigma'_m$. In particular, for f a constant in Σ_0 we get a constant f^l in $(\Sigma \otimes \Sigma')_0$.

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

The axioms of $\mathtt{T} \otimes \mathtt{T}'$ are the following:

Inherited Axioms.

For each axiom $\alpha = (\forall(x_1 : \mathtt{S}_1, \ldots, x_m : \mathtt{S}_m)\ \varphi(\vec{x}) \Leftarrow \varphi_1(\vec{x}) \ \wedge \cdots \wedge \ \varphi_n(\vec{x}))$ in Γ, with $\mathtt{S}_i = \{\mathtt{s}_{i1}, \ldots, \mathtt{s}_{il_i}\}$, for $1 \leqslant i \leqslant m$, we introduce an axiom

$$\alpha^l = \left(\forall\left(x_1 : \mathtt{S}_1^l, \ldots, x_m : \mathtt{S}_m^l\right)\ \varphi^l(\vec{x}) \Leftarrow \varphi_1^l(\vec{x}) \ \wedge \cdots \wedge \ \varphi_n^l(\vec{x})\right)$$

with $\mathtt{S}_i^l = \{(\mathtt{s}_{i1}, \top'), \ldots, (\mathtt{s}_{il_i}, \top')\}$, $1 \leqslant i \leqslant m$, and with φ^l, φ_j^l the obvious translations of φ, φ_j obtained by replacing each $f \in \Sigma$ by its corresponding f^l. Similarly, we define for each axiom $\beta \in \Gamma'$ the axiom β^r and impose all these axioms.

Subalgebra Axioms.

(i) For each $f \in \Sigma_n$ and each $\mathtt{s}' \in \mathtt{S}'$, $\mathtt{s}' \neq \top'$, we introduce the axiom

$$\forall(x_1 : (\top, \mathtt{s}'), \ldots, x_n : (\top, \mathtt{s}'))\ \ f^l(\vec{x}) : (\top, \mathtt{s}') \Leftarrow f^l(\vec{x}) : (\top, \top').$$

(ii) For each $g \in \Sigma'_m$ and each $\mathtt{s} \in \mathtt{S}$, $\mathtt{s} \neq \top$, we introduce the axiom

$$\forall(x_1 : (\mathtt{s}, \top'), \ldots, x_m : (\mathtt{s}, \top'))\ \ g^r(\vec{x}) : (\mathtt{s}, \top') \Leftarrow g^r(\vec{x}) : (\top, \top').$$

(iii) For each $(\mathtt{s}, \mathtt{s}') \in \mathtt{S} \times \mathtt{S}'$ with $\mathtt{s} \neq \top$ and $\mathtt{s}' \neq \top'$, we have the axiom

$$\forall x : (\top, \top')\ \ x : (\mathtt{s}, \mathtt{s}') \Leftarrow x : (\top, \mathtt{s}') \ \wedge \ x : (\mathtt{s}, \top').$$

Homomorphism Axioms.

For each $f \in \Sigma_n$, $g \in \Sigma'_m$, $n + m \geqslant 0$, we introduce the axiom

$$\forall \vec{x}\ f^l(g^r(\vec{x_{1_}}), \ldots, g^r(\vec{x_{n_}})) = g^r(f^l(\vec{x_{_\,1}}), \ldots, f^l(\vec{x_{_\,m}})) \Leftarrow \bigwedge_{1 \leqslant i \leqslant n} g^r(\vec{x_{i_}}) : (\top, \top') \wedge \bigwedge_{1 \leqslant j \leqslant m} f^l(\vec{x_{_\,j}}) : (\top, \top'),$$

where $\vec{x_{i_}} = \{x_{ij} : (\top, \top')\}_{1 \leqslant j \leqslant m}$, for $i \in [1, n]$, $\vec{x_{_\,j}} = \{x_{ij} : (\top, \top')\}_{1 \leqslant i \leqslant n}$, for $j \in [1, m]$, and $\vec{x} = \{x_{ij} : (\top, \top')\}_{1 \leqslant j \leqslant m}^{1 \leqslant i \leqslant n}$.

Appendix B. Theories in partial membership equational logic

As a quick reference for the reader, in this appendix we collect the partial membership equational logic theories of: categories; monoidal categories; symmetric monoidal categories, taken from Bruni *et al.* (2001); and match-share categories, taken from Bruni *et al.* (2002b).

The theory of categories `CAT` is defined as follows. Its poset of sorts has sorts `Object` and `Arrow` with `Object` < `Arrow`. There are two unary total operations `d(_)` and `c(_)`, for *domain* and *codomain*, and a binary composition operation `_;_` defined if and only if the codomain of the first argument is equal to the domain of the second argument. The complete definition of the theory is given in Figure 31. By convention, functions with given domain and codomain are total on that domain and codomain.

It is easy to check that a model of `CAT` is exactly a category (in which objects coincide with identity arrows), and that `CAT`-homomorphisms are just functors.

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

```
fth CAT is
 sorts Object Arrow.
 subsort Object < Arrow.
 ops d(_) c(_) : Arrow -> Object.
 op  _;_.
 var      a : Object.
 vars f g h : Arrow.
 eq  d(a) = a.
 eq  c(a) = a.
 ceq a;f = f   if d(f) == a.
 ceq f;a = f   if c(f) == a.
 cmb f;g : Arrow   if c(f) == d(g).
 ceq c(f) = d(g)   if f;g : Arrow.
 ceq d(f;g) = d(f)   if c(f) == d(g).
 ceq c(f;g) = c(g)   if c(f) == d(g).
 ceq (f;g);h = f;(g;h)   if c(f) == d(g) and c(g) == d(h).
endfth
```

Figure 31. The theory CAT.

```
fth MONCAT is MON ⊗ CAT renamed by (
    sort (Monoid,Object) to Object.
    sort (Monoid,Arrow) to Arrow.
    op  e left to e.
    op _⊗_ left to _⊗_.
    op _;_ right to _;_.
    op d(_) right to d(_).
    op c(_) right to c(_). ).
endfth
```

Figure 32. The theory MONCAT.

By exploiting the tensor product of theories (see Definition A.8), the theory of monoidal categories can be obtained by combining the theories MON and CAT as illustrated in Figure 32. Recall that the theory MON of monoids has a unique sort Monoid and two total operators: the associative tensor ⊗ and the unit element e, which behaves as the identity for ⊗ (see Figure 27 in Section 6).

Note that the tensor product construction MON ⊗ CAT has the sort poset originating from the product of the two sort posets in MON and CAT and operators '*opM* left' and '*opC* right' for each operator *opM* in MON and *opC* in CAT. The axioms of MON ⊗ CAT are generated by combining the axioms of MON and CAT, and can be partitioned into three classes: (A) inherited axioms; (B) subalgebra axioms; (C) homomorphism axioms. The complete definition of the theory MON ⊗ CAT is in Figure 33. The theory MONCAT just renames sorts and operators to give a more friendly notation.

The theory of symmetric monoidal categories SMONCAT is defined in Figure 34 by adding the symmetric natural transformation $\gamma(_,_)$.

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

```
fth MON ⊗ CAT is
 sorts (Monoid,Object) (Monoid,Arrow).
 subsort (Monoid,Object) < (Monoid,Arrow).
*** (B) subalgebra axioms are implicit in op declarations
 op e left : -> (Monoid,Arrow).
 op _⊗_ left : (Monoid,Arrow) (Monoid,Arrow) -> (Monoid,Arrow).
 ops d(_) right c(_) right : (Monoid,Arrow) -> (Monoid,Object).
 op  _;_ right.
 var        a : (Monoid,Object).
 vars f g h k : (Monoid,Arrow).
*** (A) inherited axioms
 eq  (f⊗g left)⊗h left = f⊗(g⊗h left) left.
 eq  f⊗(e left) left = f.
 eq  (e left)⊗f left = f.
 eq  d(a) right = a.
 eq  c(a) right = a.
 ceq a;f right = f if d(f) right == a.
 ceq f;a right = f if c(f) right == a.
 cmb f;g right : (Monoid,Arrow) if c(f) right == d(g) right.
 ceq c(f) right = d(g) right if f;g right : (Monoid,Arrow).
 ceq d(f;g right) right = d(f) right if c(f) right == d(g) right.
 ceq c(f;g right) right = c(g) right if c(f) right == d(g) right.
 ceq (f;g right);h right = f;(g;h right) right
     if c(f) right == d(g) right and c(g) right == d(h) right.
*** (C) homomorphism axioms
 eq  d(e left) right = e left.
 eq  c(e left) right = e left.
 eq  (e left);(e left) right = e left.
 eq  d(f⊗g left) right = (d(f) right)⊗(d(g) right) left.
 eq  c(f⊗g left) right = (c(f) right)⊗(c(g) right) left.
 ceq (f;g right)⊗(h;k right) left = (f⊗h left);(g⊗k left) right
     if f;g right : (Monoid,Arrow) and h;k right : (Monoid,Arrow).
endfth
```

Figure 33. Explicit definition of the theory MON ⊗ CAT.

Finally, the theory of match-share categories MSCAT is defined in Figure 35 by adding the transformations ∇(_) and Δ(_) and suitable axioms for combining them. The first group of axioms expresses the coherence of ∇, and the second that of Δ. The third group of axioms states how the two transformations can interact together. If we look at ∇(a) as a wiring establishing two connections between the object *a* in the domain and the occurrences of *a* in the codomain, and dually for Δ(a), the last two axioms say that the multiplicity of connections is not important, and that the connections work in both direction, that is, what matters is not how objects are connected, but just the fact that they are connected by some undirected path of wirings.

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

```
fth SMONCAT is including MONCAT.
 op γ(_,_) : Object Object -> Arrow.
 vars a, a', b, b', c : Object.
 vars           f, f' : Arrow.
 eq d(γ(a,b)) = a⊗b.
 eq c(γ(a,b)) = b⊗a.
 eq γ(a,e) = a.
 eq γ(e,a) = a.
 eq γ(a⊗b,c) = (a⊗γ(b,c));(γ(a,c)⊗b).
 eq γ(a,b);γ(b,a) = a⊗b.
 ceq (f⊗f');γ(b,b') = γ(a,a');(f'⊗f)
     if d(f) == a and d(f') == a' and c(f) == b and c(f') == b'.
endfth
```

Figure 34. The theory SMONCAT.

```
fth MSCAT is including SMONCAT.
 ops ∇(_) Δ(_) : Object -> Arrow.
 var a : Object.
 eq  d(∇(a)) = a.
 eq  c(∇(a)) = a⊗a.
 eq  ∇(e) = e.
 eq  ∇(a⊗b) = (∇(a)⊗∇(b));(a⊗γ(a,b)⊗b).
 eq  ∇(a);(∇(a)⊗a) = ∇(a);(a⊗∇(a)).
 eq  ∇(a);γ(a,a) = ∇(a).

 eq  d(Δ(a)) = a⊗a.
 eq  c(Δ(a)) = a.
 eq  Δ(e) = e.
 eq  Δ(a⊗b) = (a⊗γ(b,a)⊗b);(Δ(a)⊗Δ(b)).
 eq  (Δ(a)⊗a);Δ(a) = (a⊗Δ(a));Δ(a).
 eq  γ(a,a);Δ(a) = Δ(a).

 eq  ∇(a);Δ(a) = a.
 eq  Δ(a);∇(a) = (a⊗∇(a));(Δ(a)⊗a).
endfth
```

Figure 35. The theory MSCAT.

Acknowledgements

We thank the anonymous referees of ConCoord 2001 for their kind comments and suggestions on theconference version of the paper. We also want to thank an anonymous reviewer of the full version for several helpful comments and especially for spotting a technical flaw in the notation that would have compromised the results in Section 4.3. Finally, we thank Hernán Melgratti for his careful reading of the *almost final* version of this paper.

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

References

Anderson, B. and Shasha, D. (1991) Persistent Linda: Linda + Transactions + Query. In: Research Directions in High-Level Parallel Programming Languages. *Springer-Verlag Lecture Notes in Computer Science* **574** 93–109.

Asperti, A. and Busi, N. (1996) Mobile Petri nets. Technical Report UBLCS 96-10, Computer Science Department, University of Bologna. (Available at: `ftp://ftp.cs.unibo.it/pub/techreports/96-10.ps.gz`.)

Bernstein, P. A., Hadzilacos, V. and Goodman, N. (1987) *Concurrency, Control and Recovery in Database Systems*, Addison-Wesley Longman. (Available at: `http://research.microsoft.com/pubs/ccontrol`.)

Bruni, R., Laneve, C. and Montanari, U. (2002a) Orchestrating transactions in the join-calculus. In: Proc. CONCUR 2002. *Springer-Verlag Lecture Notes in Computer Science* **2421** 321–336. (Available at: `http://link.springer.de/link/service/series/0558/tocs/t2421.htm`.)

Bruni, R., Meseguer, J., Montanari, U. and Sassone, V. (2001) Functorial models for Petri nets. *Information and Computation* **170** (2) 207–236. (Available at: `http://www.idealibrary.com/links/toc/inco/170/2/0`.)

Bruni, R., Meseguer, J., Montanari, U. and Sassone, V. (2002b) Functorial models for contextual pre-nets. Technical Report TR-02-09, Computer Science Department, University of Pisa. (Available at: `ftp://ftp.di.unipi.it/pub/techreports/TR-02-09.ps.Z`.)

Bruni, R. and Montanari, U. (1997) Zero-safe nets, or transition synchronization made simple. In: Proc. EXPRESS'97. *Electronic Notes in Theoretical Computer Science*, Elsevier **7**. (Available at: `http://www.elsevier.nl/locate/entcs/volume7.html`.)

Bruni, R. and Montanari, U. (1998) Zero-safe nets: The individual token approach. In: Proc. WADT'97. *Springer-Verlag Lecture Notes in Computer Science* **1376** 122–140. (Available at: `http://www.di.unipi.it/~bruni/publications/wadt97.ps.gz`.)

Bruni, R. and Montanari, U. (2000a) Executing transactions in zero-safe nets. In: Proc. ICATPN 2000. *Springer-Verlag Lecture Notes in Computer Science* **1825** 83–102. (Available at: `http://link.springer.de/link/service/series/0558/tocs/t1825.htm`.)

Bruni, R. and Montanari, U. (2000b) Zero-safe nets: Comparing the collective and individual token approaches. *Information and Computation* **156** (1-2) 46–89. (Available at: `http://www.idealibrary.com/links/toc/inco/156/1/0`.)

Bruni, R. and Montanari, U. (2001a) Transactions and zero-safe nets. In: Advances in Petri Nets: Unifying Petri Nets. *Springer-Verlag Lecture Notes in Computer Science* **2128** 380–426. (Available at: `http://link.springer.de/link/service/series/0558/tocs/t2128.htm`.)

Bruni, R. and Montanari, U. (2001b) Zero-safe net models for transactions in Linda. In: Proc. ConCoord 2001. *Electronic Notes in Theoretical Computer Science*, Elsevier **54**. (Available at: `http:// www.elsevier.nl/locate/entcs/volume54.html`.)

Bruni, R. and Sassone, V. (2000) Algebraic models for contextual nets. In: Proc. ICALP 2000. *Springer-Verlag Lecture Notes in Computer Science* **1853** 175–186. (Available at: `http://link.springer.de/link/service/series/0558/tocs/t1853.htm`.)

Bruni, R. and Sassone, V. (2001) Two algebraic process semantics for contextual nets. In: Advances in Petri Nets: Unifying Petri Nets. *Springer-Verlag Lecture Notes in Computer Science* **2128** 427–456. (Available at: `http://link.springer.de/link/service/series/0558/tocs/t2128.htm`.)

Buscemi, M. and Sassone, V. (2001) High-level Petri nets as type theories in the Join calculus. In: Proc. FoSSaCS'01. *Springer-Verlag Lecture Notes in Computer Science* **2030** 104–120. (Available at: `http://link.springer.de/link/service/series/0558/tocs/t2030.htm`.)

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

Busi, N. (1998) *Petri Nets with Inhibitor and Read Arcs: Semantics, Analysis and Application to Process Calculi*, Ph.D. thesis, Mathematics Department, University of Siena. (Available at: `http://www.cs.unibo.it/~busi/www/thesis.ps.gz`.)

Busi, N., Gorrieri, R. and Zavattaro, G. (1996) A truly concurrent view of Linda interprocess communication. Technical Report UBLCS 97-02, Computer Science Dept., Univ. of Bologna. (Available at: `ftp://ftp.cs.unibo.it/pub/techreports/97-02.ps.gz`.)

Busi, N., Gorrieri, R. and Zavattaro, G. (1998) A process algebraic view of Linda coordination primitives. *Theoretical Computer Science* **192** (2) 167–199.

Busi, N., Gorrieri, R. and Zavattaro, G. (2000a) Comparing three semantics for Linda-like languages. *Theoretical Computer Science* **240** (1) 49–90.

Busi, N., Gorrieri, R. and Zavattaro, G. (2000b) On the expressiveness of Linda coordination primitives. *Information and Computation* **156** (1–2) 90–121. (Available at: `http://www.idealibrary.com/links/toc/inco/156/1/0`.)

Busi, N., Gorrieri, R. and Zavattaro, G. (2000c) Process calculi for coordination: from Linda to JavaSpaces. In: Proc. AMAST'00. *Springer-Verlag Lecture Notes in Computer Science* **1816** 198–212. (Available at: `http://link.springer.de/link/service/series/0558/tocs/t1816.htm`.)

Busi, N. and Zavattaro, G. (2001) On the serializability of transactions in JavaSpaces. In: Proc. ConCoord 2001. *Electronic Notes in Theoretical Computer Science*, Elsevier **54**. (Available at: `http://www.elsevier.nl/locate/entcs/volume54.html`.)

Christensen, S. and Hansen, N. D. (1993) Coloured Petri nets extended with place capacities, test arcs and inhibitor arcs. In: ICATPN'93. *Springer-Verlag Lecture Notes in Computer Science* **691** 186–205.

De Francesco, N., Montanari, U. and Ristori, G. (1994) Modeling concurrent accesses to shared data via Petri nets. Programming Concepts, Methods and Calculi. *IFIP Transactions*, North Holland **A-56** 403–422.

De Nicola, R., Ferrari, G. and Pugliese, R. (1998) KLAIM: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, IEEE Computer Society **24** (5) 315–330.

Desel, J., Juhás, G. and Lorenz, R. (2001) Petri Nets over Partial Algebra. In: Advances in Petri Nets: Unifying Petri Nets. *Springer-Verlag Lecture Notes in Computer Science* **2128** 126–172. (Available at: `http://link.springer.de/link/service/series/0558/tocs/t2128.htm`.)

Fournet, C., Gonthier, G., Lévy, J.-J., Maranget, L. and Rémy, D. (1996) A calculus of mobile agents. In: Proc. CONCUR'96. *Springer-Verlag Lecture Notes in Computer Science* **1119** 406–421.

Gadducci, F. and Montanari, U. (1998) Axioms for contextual net processes. In: Proc. ICALP'98. *Springer-Verlag Lecture Notes in Computer Science* **1443** 296–308. (Available at: `http://link.springer.de/link/service/series/0558/tocs/t1443.htm`.)

Gelernter, D. (1985) Generative communication in Linda. *ACM Transactions on Programming Languages and Systems* **7** (1) 80–112.

van Glabbeek, R. J. and Plotkin, G. D. (1995) Configuration structures. *Proc. LICS'95*, IEEE Computer Society Press 199–209. .

Groote, J. F. (1993) Transition system specifications with negative premises. *Theoretical Computer Science* **118** 263–299.

Janicki, R. and Koutny, M. (1995) Semantics of inhibitor nets. *Information and Computation* **123** (1) 1–16. (Available at: `http://www.idealibrary.com/links/toc/inco/123/1/0`.)

Meseguer, J. (1998) Membership algebra as a logical framework for equational specification. In: Proc. WADT'97. *Springer-Verlag Lecture Notes in Computer Science* **1376** 18–61.

Meseguer, J. and Montanari, U. (1990) Petri Nets are Monoids. *Information and Computation* **88** (2) 105–155.

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press

Montanari, U. and Rossi, F. (1994) Contextual occurrence nets and concurrent constraint programming. In: Graph Transformations in Computer Science. *Springer-Verlag Lecture Notes in Computer Science* **776** 280–295.

Montanari, U. and Rossi, F. (1995) Contextual nets. *Acta Informatica* **32** 545–596.

Sun Microsystems, Inc. (2000) JavaSpaces™ specifications, v.1.1. (Available at: `http://www.sun.com/jini/specs/js1_1.pdf`.)

Vogler, W. (1997a) Partial order semantics and read arcs. In: Proc. MFCS'97. *Springer-Verlag Lecture Notes in Computer Science* **1295** 508–517.

Vogler, W. (1997b) Efficiency of asynchronous systems and read arcs in Petri nets. In: Proc. ICALP'97. *Springer-Verlag Lecture Notes in Computer Science* **1256** 538–548.

Wyckoff, P., McLaughry, S. W., Lehman, T. J. and Ford, D. A. (1998) T Spaces. *IBM Systems Journal* **37** (3) 454–474. (Available at: `http://www.research.ibm.com/journal/sj/373/wyckoff.html`.)

https://doi.org/10.1017/S0960129504004189 Published online by Cambridge University Press