9-2-2004

The Logical Meeting Point of Multiset Rewriting and Process Algebra: Progress Report

Iliano Cervesato
ITT Industries

The Logical Meeting Point of Multiset Rewriting and Process Algebra: *Progress Report*

Iliano Cervesato *

Advanced Engineering and Science Division, ITT Industries Inc.
Alexandria, VA 22303, USA
iliano@itd.nrl.navy.mil

Technical Memo 5540-153
Center for High Assurance Computer Systems
Naval Research Laboratory — Washington, DC
September 2, 2004

Abstract. We present a revisited semantics for multiset rewriting founded on the left sequent rules of linear logic in its LV presentation. The resulting interpretation is extended with a majority of linear connectives into the language of ω-*multisets*. It drops the distinction between multiset elements and rewrite rules, and considerably enriches the expressive power of standard multiset rewriting with embedded rules, choice, replication and more. The cut rules introduce finite auxiliary rewriting chains and are admissible. Derivations are now primarily viewed as open objects, and are closed only to examine intermediate rewriting states. The resulting language can also be interpreted as a process algebra. A simple translation maps process constructors of the asynchronous π-calculus to rewrite operators, while the structural equivalence corresponds directly to logically-motivated structural properties of ω-multisets (with one exception). The language of ω-multisets forms the basis for the security protocol specification language MSR 3. With relations to both multiset rewriting and process algebra, it supports specifications that are process-based, state-based, or of a mixed nature. Additionally, its deep logical underpinning makes it an ideal common ground for systematically comparing protocol specification languages, a task currently done in an ad-hoc manner.

Keywords: Linear logic, multiset rewriting, process algebra, security protocols.

1 Introduction

The semantics of a logic is generally given as a set of inference rules that can be composed to build derivations. Traditionally, derivations are used to support judgments such as the entailment of a formula from given assumptions. To this end, a derivation shall

* Partially supported by NRL under contract N00173-00-C-2086. This research was conducted while the author was visiting Princeton University.

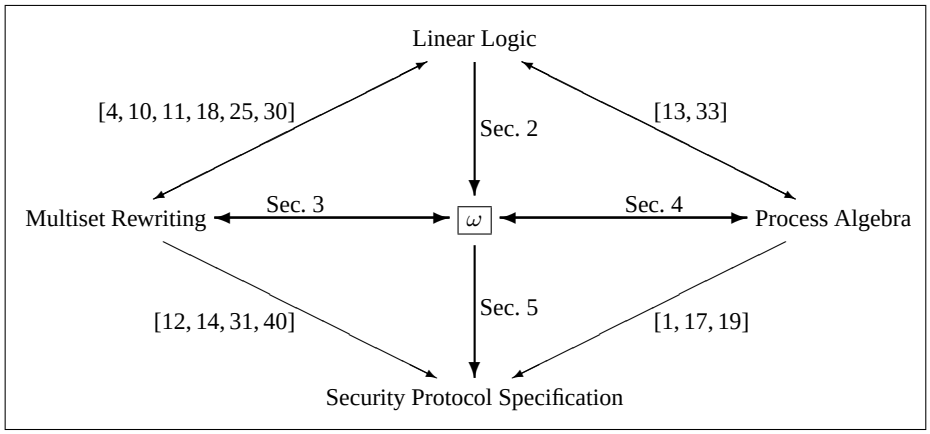

Fig. 1. Overview

be finite and closed, in the sense that the premises of every rule in it are themselves justified by (sub-)derivations.

In this paper, we emphasize a radically different view of rules, derivations, and ultimately logic. We will be primarily interested in the vertical process of extending open derivations upwards, with little concern for finiteness. The horizontal process of closing a derivation (and proving something, in the traditional sense) will be of secondary importance, mostly as a form of observation.

We develop this idea with respect to a fragment of intuitionistic linear logic [24] in Pfenning's LV sequent presentation [42]. We turn LV's left rules into a form of rewriting over logical contexts. It transforms a rule's conclusion into its major premise, with minor premises corresponding to finite auxiliary rewriting chains (they can be in-lined using the cut rules). The axiom rule becomes a means of observing the rewriting process. A few of LV's right rules indirectly contribute to a notion of equivalence, while the rest is discarded. It is shown that LV's cut rules are admissible.

The resulting system, which we call ω, is much weaker than LV (because of the absence of right rules), but constitute a powerful form of rewriting. We show that a tiny syntactic fragment of ω corresponds exactly to traditional multiset rewriting (or place/transition Petri nets). This constitutes an interpretation of multiset rewriting *as* (a fragment of) logic, which we like to contrast to the previous interpretations *into* (a fragment of) logic [4, 10, 11, 18, 25, 30]. The system ω similarly provides a new logical foundation to more sophisticated forms of multiset rewriting and Petri nets.

Pushing this methodology further, we view ω as an extreme form of multiset rewriting: it drops the distinction between multiset elements and rewrite rules, and considerably enriches the expressive power of standard multiset rewriting with embedded rules, parametricity, choice, replication and more. Yet, its semantics is given by the rules of logic. Under this interpretation, we call formulas ω-multisets.

The system ω has also close ties to process algebra, in particular to the join calculus [23] and the asynchronous π-calculus [38, 44]. A simple execution-preserving translation maps process constructors of the latter to rewrite operators, while its struc-

tural equivalence corresponds directly to logically motivated properties of ω (with one exception).

With relations to the two major paradigms for distributed and concurrent computing, ω is a promising middle ground where both state-based and process-based specifications can coexist. We test this proposition in the arena of cryptographic protocol analysis, in which both approaches are prominently used, and only ad-hoc mappings exist to bridge them. We develop ω into the protocol specification language MSR 3 and scrutinize various ways of expressing a protocol.

This paper is organized as follows. Section 2 distills ω out of LV. Section 3 exposes ω as a new form of multiset rewriting. Section 4 relates it to the process algebraic world. Section 5 brings the two together in the applied domain of security protocols.[1] Additional remarks and ideas for future developments are given in Section 6. Figure 1 summarizes the functional relations between the various languages touched in this paper, as found in the literature (along the thin edges) and in the present work (along the thick edges).

2 A Rewriting View of Linear Logic

In this section, we will give a rewriting interpretation to a fragment of linear logic in its LV sequent presentation. We then refine it by cut-elimination into a system that we call ω.

2.1 LV Sequents

We base our investigation on the following fragment of intuitionistic linear logic [24]:

$$A, B, C \ ::= \ a \mid \mathbf{1} \mid A \otimes B \mid A \multimap B \mid !A \mid \top \mid A \,\&\, B \mid \forall x.\, A \mid \exists x.\, A$$

Here, a and x range over atomic formulas and variables, respectively. We do not distinguish formulas that differ only by the name of their bound variables, and rely on implicit α-renaming whenever convenient. We write $[t/x]A$ for the capture avoiding substitution of term t for x in A, and $\mathrm{FV}(A)$ for the set of free variables occurring in A. We shall not place any restriction on the embedded term language except for predicativity (term substitution cannot alter the outer structure of a formula). However, the applications in this paper will only require a first-order term language (extended with sorts in Section 5). While we limit our attention to the listed operators of linear logic, we will comment on other connectives in Section 2.4.

Our definition of provability is based on an intuitionistic version of Pfenning's LV sequent calculus [42]. It relies on sequents of the form

$$\Gamma; \Delta \ \longrightarrow_{\Sigma} \ C.$$

[1] For the chronicle, this research developed almost opposite to this narration: while relating multiset rewriting and process algebraic languages for security protocols, we considered an extension to the former with embedded rewrite rules. This led to noticing the relation to the treatment of contexts in the sequent calculus presentation of linear logic. Formalizing this aspect yielded the structural properties, and the observation that they correspond almost exactly to the structural equivalences of the π-calculus.

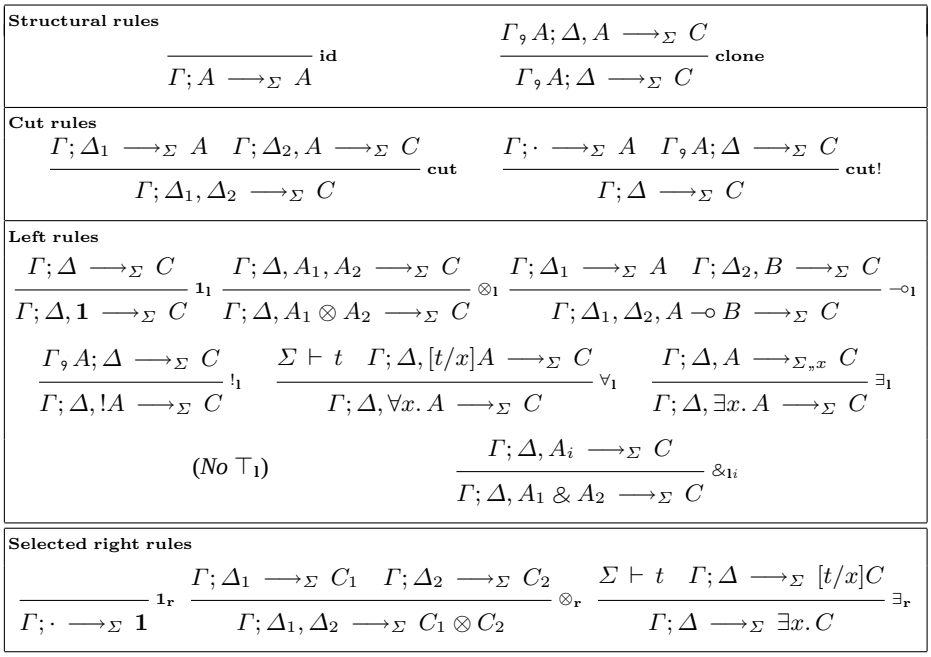

Fig. 2. LV Sequent Presentation of Intuitionistic Linear Logic

Similarly to Barber's DILL [6] and Hodas and Miller's $\mathcal{L}$ [26], LV isolates reusable assumptions in the *unrestricted context* Γ (subject to exchange, weakening and contraction), while assumptions to be used exactly once are contained in the *linear context* Δ (subject only to exchange). The combination corresponds to the single context $(!\Gamma, \Delta)$ of [24]. The *signature* Σ lists the term-level symbols in use. We call C the *goal formula*.

We shall be very precise when discussing the structure of contexts and signatures. Therefore, we will use different symbols for their constructors, as given by the following grammar:

$$\begin{array}{rcl} \Delta & ::= & \cdot \mid \Delta, A \\ \Gamma & ::= & \circ \mid \Gamma, A \\ \Sigma & ::= & \text{-} \mid \Sigma, x \end{array}$$

For each of these collections, the comma (",", ",", ",") stands for the extension operator while the bullet ("$\cdot$", "$\circ$", "-") represents the empty collection. The former will be overloaded into a union operator. From an algebraic perspective, signatures, linear and unrestricted contexts will be commutative monoids. Additionally, signatures shall not contain duplicate symbols (we will extend them only with eigenvariables and rely on α-renaming to ensure this constraint).

Given these conventions, Figure 2 presents an intuitionistic subset of the sequent rules for LV [42]. The first segment contains the axiom rule (**id**) and rule **clone** that allows repeatedly using an unrestricted assumption in a derivation. The second segment lists the two applicable cut rules of LV. The left sequent rules for the fragment consid-

$\boxed{\begin{array}{l}\Delta \quad \equiv \bigotimes \Delta \\ \Sigma; \Delta \equiv \exists \Sigma.\, \Delta\end{array}}$	Assoc. :	$A \otimes (B \otimes C) \equiv (A \otimes B) \otimes C$	
	Unit :	$A \otimes \mathbf{1} \equiv A$	
	Comm. :	$A \otimes B \equiv B \otimes A$	
	assoc. :	$\exists x.\, (A \otimes B) \equiv A \otimes \exists x.\, B$	if $x \notin \mathrm{FV}(A)$
	unit :	$\exists x.\, \mathbf{1} \equiv \mathbf{1}$	
	comm. :	$\exists x.\, \exists y.\, A \equiv \exists y.\, \exists x.\, A$	
id :	$\Sigma; \Gamma; \Delta \Rrightarrow^* \Sigma; \Delta$		
Trans. :	$\Sigma; \Gamma; \Delta \Rrightarrow^* \Sigma''; \Delta''$	if	$\Sigma; \Gamma; \Delta \Rightarrow \Sigma'; \Gamma'; \Delta'$ and $\Sigma'; \Gamma'; \Delta' \Rrightarrow^* \Sigma''; \Delta''$
clone :	$\Sigma; (\Gamma, A); \Delta \Rightarrow \Sigma; (\Gamma, A); (\Delta, A)$		
cut :	$\Sigma; \Gamma; (\Delta_1, \Delta_2) \Rightarrow \Sigma; \Gamma; (\Delta_2, A)$	if	$\Sigma; \Gamma; \Delta_1 \Rrightarrow^* \Sigma; A$
cut! :	$\Sigma; \Gamma; \Delta \Rightarrow \Sigma; (\Gamma, A); \Delta$	if	$\Sigma; \Gamma; \cdot \Rrightarrow^* \Sigma; A$
$\mathbf{1}_\mathrm{l}$:	$\Sigma; \Gamma; (\Delta, \mathbf{1}) \Rightarrow \Sigma; \Gamma; \Delta$		
$\otimes_\mathrm{l}$:	$\Sigma; \Gamma; (\Delta, A_1 \otimes A_2) \Rightarrow \Sigma; \Gamma; (\Delta, A_1, A_2)$		
$\multimap_\mathrm{l}$:	$\Sigma; \Gamma; (\Delta_1, \Delta_2, A \multimap B) \Rightarrow \Sigma; \Gamma; (\Delta_2, B)$	if	$\Sigma; \Gamma; \Delta_1 \Rrightarrow^* \Sigma; A$
$\forall_\mathrm{l}$:	$\Sigma; \Gamma; (\Delta, \forall x.\, A) \Rightarrow \Sigma; \Gamma; (\Delta, [t/x]A)$	if	$\Sigma \vdash t$
$\exists_\mathrm{l}$:	$\Sigma; \Gamma; (\Delta, \exists x.\, A) \Rightarrow (\Sigma, x); \Gamma; (\Delta, A)$		
$(\top_\mathrm{l})$:	*(No rule for $\top$)*		
$\&_{\mathrm{l}i}$:	$\Sigma; \Gamma; (\Delta, A_1 \,\&\, A_2) \Rightarrow \Sigma; \Gamma; (\Delta, A_i)$		
$!_\mathrm{l}$:	$\Sigma; \Gamma; (\Delta, !A) \Rightarrow \Sigma; (\Gamma, A); \Delta$		

Fig. 3. A Rewriting Interpretation of LV

ered above are listed next. Observe how !'ed linear assumption are made available in the unrestricted context in rule $!_\mathrm{l}$. In rule $\forall_\mathrm{l}$, we rely on the auxiliary judgment $\Sigma \vdash t$ to ascertain that the term t is valid with respect to signature Σ (but do not define this notion further).

Whenever one of these rules has premises, one of them mentions the same goal formula (systematically written C) as the rule's conclusion. We will call it the *major premise* of the rule. The cut rules and $\multimap_\mathrm{l}$ also have a *minor premise* in which the goal formula changes.

The right sequent rules of linear logic have marginal importance in this work. The bottom part of Figure 2 lists some of them, as they will play an indirect role in the development. It is conceivable, however, that these and other right rules can be useful query tools, as demonstrated for example in [18, 25] relative to Petri nets. This however goes beyond the scope of this work.

Derivations are defined as usual, and denoted $\mathcal{D}$. A partial derivation $\mathcal{D}[]$ missing justification for exactly one sequent is *incomplete*. $\mathcal{D}[]$ called *open* if it is incomplete along a path from the end-sequent that only follows the major premises of the rules.

2.2 A Rewriting Interpretation of LV

With the exception of $\mathbf{id}$, the rules in the three upper segments of Figure 2 can be interpreted as a transformation of the sequent in their conclusion to the sequent in their major premise, possibly subject to side-conditions given by a minor premise. We formalize this observation as a rewrite system whose *states* are triples $(\Sigma; \Gamma; \Delta)$ consisting of the signature and the two contexts of an LV sequent. We deliberately omit the goal formula (C) for two reasons: technically, it never changes going from the conclusion to the major premise of a rule; strategically, we embrace this as an opportunity to explore logical derivations as open-ended processes rather than finite justifications of the provability of a goal given a priori. We denote this form of upward step in a derivation by means of the rewrite judgment

$$\Sigma; \Gamma; \Delta \;\Rightarrow\; \Sigma'; \Gamma'; \Delta'$$

reserving the form $\_ \;\Rightarrow^*\; \_$ for its reflexive and transitive closure. Our progresses can be tracked on Figure 3.

Given this interpretation, we can regard the minor premise in rules $\mathbf{cut}$, $\mathbf{cut}!$ and $\multimap_{\mathbf{l}}$ as prescribing the existence of an auxiliary finite rewriting chain that enables the step associated to each of these rules (the judgment $\Sigma \vdash t$ in rule $\forall_{\mathbf{l}}$ is instead a simple side-condition). Consolidating this intuition requires introducing some extra machinery. First, note that the subderivation corresponding to this auxiliary chain must be finite, and therefore is capped by a rule without premises, often $\mathbf{id}$. This implements a shift of focus from the left-hand side of a sequent to its right-hand side. We interpret this as an observation.

We will be interested in observing the contents of the linear context Δ of an arbitrary state $(\Sigma; \Gamma; \Delta)$. In order to maintain a precise accounting of the symbols in use, we define the *observation* of $(\Sigma; \Gamma; \Delta)$ as the pair $(\Sigma; \Delta)$.[2] Making an observation can then be expressed by the judgment

$$\Sigma; \Gamma; \Delta \;\Rrightarrow\; \Sigma; \Delta.$$

Notice that it differs from the axiom rule $\mathbf{id}$ only because the linear context Δ can be arbitrary rather than a single formula A. We produce an exact correspondence by identifying contexts and formulas, an idea familiar from categorical interpretations of logic [45]. More precisely, we identify the tensor $\otimes$ and its unit $\mathbf{1}$ with the union "," and unit "$\cdot$" constructors of linear contexts, respectively. Therefore, a linear context Δ is interpreted as the formula $\bigotimes\Delta$ obtained by tensoring together all its constituent formulas. This is the essence of the symmetric monoidal (closed) structure that underlies most categorical models of linear logic [45]. From a sequent calculus point of view, this is acceptable since $\Gamma; \Delta \longrightarrow_\Sigma C$ has a derivation if and only if $\Gamma; \bigotimes\Delta \longrightarrow_\Sigma C$ has one.[3]

[2] The investigation of a notion of observation that includes the unrestricted context Γ is left for future work.

[3] A proof of the forward direction only uses $\otimes_{\mathbf{l}}$ and possibly $\mathbf{1}_{\mathbf{l}}$. The reverse direction relies on $\mathbf{cut}$ and the sequent $\Gamma; \Delta \longrightarrow_\Sigma \bigotimes\Delta$ whose simple derivation uses rules $\mathbf{id}$, $\otimes_{\mathbf{r}}$ and $\mathbf{1}_{\mathbf{r}}$; $\mathbf{cut}$ can later be eliminated.

From now on, we will use $\otimes$ and "," interchangeably (and similarly for $\mathbf{1}$ and "$\cdot$"). For the ease of the reader, we will tend to prefer $\otimes$ and $\mathbf{1}$ within the scope of other logical operators and in observation states, while "," and "$\cdot$" will appear at the top level of a regular state. We shall stress, however, that they are now only notational variants for the same concept.

The algebraic properties of linear contexts as commutative monoids can then be written as explicit *structural laws* under the logical interpretation:

$$\begin{array}{lrcl} \text{Assoc.} & : A \otimes (B \otimes C) & \equiv & (A \otimes B) \otimes C \\ \text{Unit} & : \quad A \otimes \mathbf{1} & \equiv & A \\ \text{Comm.} & : \quad A \otimes B & \equiv & B \otimes A \end{array}$$

These identities over linear contexts correspond to the notion of logical equivalence given by inter-derivability, *i.e.*, $A_1 \equiv A_2$ iff for all Σ, Γ, there are derivations for both $\Gamma; A_1 \longrightarrow_\Sigma A_2$ and $\Gamma; A_2 \longrightarrow_\Sigma A_1$.

Observe that it would be incorrect to similarly fold the unrestricted context constructors $,$ and $\circ$ into $\otimes$ and $\mathbf{1}$ since $!(A \otimes B)$ is not equivalent to $!A \otimes !B$ in linear logic. This is our main reason for choosing different notations for their constructors.

Going back to our goal of interpreting the subderivations originating on a minor premise as auxiliary rewrite chains, we define the multi-step observation judgment

$$\Sigma; \Gamma; \Delta \Rrightarrow^* \Sigma^\star; \Delta^\star$$

as the composition of $\_ \Rrightarrow^* \_$ and $\_ \Rrightarrow \_$, or more directly:

$$\begin{array}{lll} \mathbf{id} & : \Sigma; \Gamma; \Delta \Rrightarrow^* \Sigma; \Delta & \\ \text{Trans.} & : \Sigma; \Gamma; \Delta \Rrightarrow^* \Sigma^\star; \Delta^\star & \text{if } \Sigma; \Gamma; \Delta \Rightarrow \Sigma'; \Gamma'; \Delta' \\ & & \text{and } \Sigma'; \Gamma'; \Delta' \Rrightarrow^* \Sigma^\star; \Delta^\star \end{array}$$

Were it not for $\exists$, this would constitute an adequate rewriting interpretation of LV. To visualize the remaining issue, consider for example a derivation $\mathcal{D}$ of the minor premise $\Gamma; \Delta_1 \longrightarrow_\Sigma A$ of rule $\mathbf{cut}$:

$$\begin{array}{ccc} \overline{\Gamma, \Gamma'; A' \longrightarrow_{\Sigma, \Sigma'} A'} & & \\ \nwarrow \quad \vdots \quad \swarrow & & \\ \Gamma; \Delta_1 \longrightarrow_\Sigma A & & \end{array}$$

By the time a branch of $\mathcal{D}$ is closed, for example by rule $\mathbf{id}$ in this sketch, uses of rule $\exists_\mathbf{l}$ will have extended the original signature Σ with new symbols Σ' (rule $!_\mathbf{l}$ will have similarly extended Γ, but this is of little concern to us). The formula A' in this instance of $\mathbf{id}$ may mention symbols x_i in Σ', and may also contribute to the overall goal formula A. Now, since A is defined over Σ, the noted uses of x_i in A' must occur bound in A. With Figure 2 as our definition of provability, this binder is $\exists$ and rule $\exists_\mathbf{r}$ has introduced it.[4]

[4] Of course, $\forall$ is as likely a candidate in a complete proof system for linear logic. Our objective is not completeness, however, and this discussion should be taken as motivation only.

With this understanding of derivations as a guideline, we identify observation states $\Sigma; \Delta$ and existential formulas $\exists\Sigma.\, \Delta$, seen as an abbreviation for $\exists x_1.\, \ldots \exists x_n.\, \bigotimes \Delta$ where $\Sigma = (x_1, \ldots, x_n)$. This is logically justified by the fact that, if $\Gamma; \Delta \longrightarrow_\Sigma C$ is derivable, so is $\circ; \exists\Sigma.\, (!\Gamma \otimes \Delta) \longrightarrow.\ \exists\Sigma.\, C$ where $!\Gamma \otimes \Delta = \bigotimes_{A \text{ in } \Gamma} !A \otimes \bigotimes \Delta$.[5] This technique is reminiscent of the notion of "telescope" in the AUTOMATH languages [48]. It also appears in recent work on concurrent constraint programming [20].

Having further blurred the distinction between the brick and mortar of sequents (or states) and the logical operators, we will use the notations $\Sigma; \Delta$ and $\exists\Sigma.\, \Delta$ interchangeably, often mixing them as in following sketch of the rewrite chain corresponding to a derivation of the minor premise $\Gamma; \Delta_1 \longrightarrow_\Sigma A$ of the hypothetical use of $\mathbf{cut}$ above:

$$\begin{array}{ccc} (\Sigma, \Sigma'); (\Gamma, \Gamma'); \Delta^\star & \Rrightarrow & (\Sigma, \Sigma'); \Delta^\star \\ {\scriptstyle *}\Uparrow & & \| \\ \Sigma; \Gamma; \Delta_1 & \Rightarrow^* & \Sigma; \underbrace{\exists\Sigma'.\, \Delta^\star}_{A} \end{array}$$

Here, we fold the added symbols Σ' into the observed linear context $\Delta^\star$ in order to construct the formula used in the major premise. Identical considerations apply to any LV rule with a minor premise (here $\mathbf{cut}!$ and $\multimap_{\mathbf{l}}$).

With the rewrite steps induced by rules $\mathbf{cut}$, $\mathbf{cut}!$ and $\multimap_{\mathbf{l}}$ ending in states of the form $\Sigma; \Gamma; (\Delta, A)$ with $A = \exists\Sigma'.\, \Delta'$, it is natural to allow individual binders $\exists x$ among $\exists\Sigma'$ to move around, either to hug more closely formulas in Δ' or to extend their scope to include elements of Δ, as long as this does not cause either bound symbols to become free or variable capture. We formalize this possibility by means of the following *mobility laws*, which extend the monoidal equivalence $\equiv$ introduced earlier:

$$\begin{array}{lrcl} \text{assoc.}: & \exists x.\, (A \otimes B) & \equiv & A \otimes \exists x.\, B \text{ if } x \notin \mathrm{FV}(A) \\ \text{unit} \quad : & \exists x.\, \mathbf{1} & \equiv & \mathbf{1} \\ \text{comm.}: & \exists x.\, \exists y.\, A & \equiv & \exists y.\, \exists x.\, A \end{array}$$

The first pushes binders inside a formula (or state) by skipping objects where it does not occur, the second eliminates unused binders, and the third allows binders to commute. As for the monoidal laws, the formulas on each side of $\equiv$ are inter-derivable in linear logic. Notice the resemblance between the monoidal and mobility laws (highlighted through related labels), that type theory explains by pointing out that an existential quantifier can be interpreted as a form of dependent conjunction.

This completes our rewriting interpretation of LV. The resulting rewrite rules are summarized in Figure 3. With the exception of the added rule Tran, each maintains the name of the LV inference it was obtained from. Rules $\mathbf{1}_{\mathbf{l}}$ and $\otimes_{\mathbf{l}}$ have been grayed out as redundant since they are subsumed by the identification of linear contexts and tensored formulas. Rules $\mathbf{cut}$, $\mathbf{cut}!$ and $\multimap_{\mathbf{l}}$ make implicit use of the identification of observation states and existential formulas, just described.

[5] This proof extends the technique seen in the forward direction of Footnote 3 with uses of $\exists_{\mathbf{l}}$, $\exists_{\mathbf{r}}$ and $!_{\mathbf{l}}$, in addition to $\otimes_{\mathbf{l}}$ and $\mathbf{1}_{\mathbf{l}}$. Notice that $\exists_{\mathbf{r}}$ invariably uses a variable x in Σ as the substitution term t, giving $\exists$ a flavor very close to Miller and Tiu's ∇ [37]. The reverse of this property does not hold in general. With some surprise, we could not find a categorical counterpart of this technique.

id	:	$\Sigma; \Gamma; \Delta \Rrightarrow^* \Sigma; \Delta$
Trans.	:	$\Sigma; \Gamma; \Delta \Rrightarrow^* \Sigma''; \Delta''$ if $\Sigma; \Gamma; \Delta \Rightarrow \Sigma'; \Gamma'; \Delta'$ and $\Sigma'; \Gamma'; \Delta' \Rrightarrow^* \Sigma''; \Delta''$
clone	:	$\Sigma; (\Gamma, A); \Delta \Rightarrow \Sigma; (\Gamma, A); (\Delta, A)$
$\multimap_1'$	:	$\Sigma; \Gamma; (\Delta, A, A \multimap B) \Rightarrow \Sigma; \Gamma; (\Delta, B)$
$\forall_1$	:	$\Sigma; \Gamma; (\Delta, \forall x.\, A) \Rightarrow \Sigma; \Gamma; (\Delta, [t/x]A)$ if $\Sigma \vdash t$
$\exists_1$	:	$\Sigma; \Gamma; (\Delta, \exists x.\, A) \Rightarrow (\Sigma, x); \Gamma; (\Delta, A)$
$\&_{1i}$	:	$\Sigma; \Gamma; (\Delta, A_1 \,\&\, A_2) \Rightarrow \Sigma; \Gamma; (\Delta, A_i)$
$!_1$	:	$\Sigma; \Gamma; (\Delta, !A) \Rightarrow \Sigma; (\Gamma, A); \Delta$

Fig. 4. The Rules of ω-Rewriting

The rewriting interpretation displayed in Figure 3 is sound with respect to the rules of linear logic, even when extended with the right rules not considered in Figure 2. This is expressed by the following property:

Property 1 (Soundness).

1. If $\Sigma; \Gamma; \Delta \Rightarrow^* \Sigma'; \Gamma'; \Delta'$, then there exist LV formulas C and C' and a derivation $\mathcal{D}[]$ of $\Gamma; \Delta \longrightarrow_\Sigma C$ open at $\Gamma'; \Delta' \longrightarrow_{\Sigma'} C'$.
2. If $\Sigma; \Gamma; \Delta \Rrightarrow^* (\Sigma, \Sigma'); \Delta^\star$, then there is an LV derivation $\mathcal{D}$ of $\Gamma; \Delta \longrightarrow_\Sigma \exists \Sigma'. \bigotimes \Delta^\star$.

Clearly, no completeness result holds as we have forsaken most right rules of LV: for example, no rewrite chain can validate $\cdot; \circ; (a \multimap b, b \multimap c) \Rrightarrow^* \cdot; (a \multimap c)$, although linear implication *is* transitive in linear.

2.3 Cut-Elimination and ω-Rewriting

The interpretation of LV as a rewrite system in Figure 3 is unusual in the sense that the single step relation $\Rightarrow$ depends on the multi-step observation relation $\Rrightarrow^*$ in rules $\multimap_1$, **cut** and **cut**!. In this section, we refine it into a presentation that is immune from this oddity. We will call it system ω, and refer to its use as ω-rewriting.

First observe that rule $\multimap_1$ admits the simplified form

$$\multimap_1' \;:\quad \Sigma; \Gamma; (\Delta, A, A \multimap B) \Rightarrow \Sigma; \Gamma; (\Delta, B).$$

Indeed, every use of $\multimap_1$ in a rewrite sequence can be replaced with an instance of **cut** followed by one of $\multimap_1'$.

More interestingly, like their logical counterparts [42], both cut rules are admissible in our rewrite system, *i.e.*, any rewriting sequence can be transformed into an observationally equivalent *cut-free* sequence that does not make use of them. Intuitively, this will amount to in-lining the auxiliary rewriting chain whenever one of the cut rules is used. A formal account follows the lines of a standard proof of cut-elimination, *e.g.* [42], but is not as involved.

We first prove the following weakening lemma by a simple induction on the given rewriting sequence.

Lemma 1 (Weakening).
For any Σ', Γ' and Δ', if $\Sigma; \Gamma; \Delta \Rrightarrow^* \Sigma^\star; \Delta^\star$, then $(\Sigma \varsigma \Sigma'); (\Gamma \varsigma \Gamma'); (\Delta, \Delta') \Rrightarrow^* (\Sigma^\star \varsigma \Sigma'); (\Delta^\star, \Delta')$.

The most delicate aspect of the work in this section is the proper accounting for signature symbols. This can be summarized in the following lemma, also proved by induction.

Lemma 2.
If $\Sigma; \Gamma; \exists \Sigma'. \Delta \Rrightarrow^* \Sigma^\star; \Delta^\star$, then $(\Sigma, \Sigma'); \Gamma; \Delta \Rrightarrow^* \Sigma^\star; \Delta^\star$.

We can now tackle the rewriting equivalent the admissibility of the cut rules that is, every chain that could be produced from two cut-free chains by means of $\mathbf{cut}$ (or $\mathbf{cut}!$), can also be obtained without.

Lemma 3 (Admissibility of $\mathbf{cut}$ and $\mathbf{cut}!$).

1. *For any cut-free rewriting chains $\Sigma; \Gamma; \Delta_1 \Rrightarrow^* \Sigma; A$ and $\Sigma; \Gamma; (\Delta_2, A) \Rrightarrow^* \Sigma^\star; \Delta^\star$, there is a cut-free sequence $\Sigma; \Gamma; (\Delta_1, \Delta_2) \Rrightarrow^* \Sigma^\star; \Delta^\star$.*
2. *For any cut-free rewriting chains $\Sigma; \Gamma; \cdot \Rrightarrow^* \Sigma; A$ and $\Sigma; (\Gamma \varsigma A); \Delta \Rrightarrow^* \Sigma^\star; \Delta^\star$, there is a cut-free sequence $\Sigma; \Gamma; \Delta \Rrightarrow^* \Sigma^\star; \Delta^\star$.*

The proof of (1) simply prefixes the first rewriting chain to the second, using the above lemmas to align signatures and contexts. As for (2), we shall replace every application of rule $\mathbf{clone}$ on the formula A with a similar in-lining of the first rewriting chain.

On the basis of this lemma, we can eliminate every occurrence of $\mathbf{cut}$ and $\mathbf{cut}!$ in a rewriting chain.

Theorem 1 (Cut elimination).
For every rewrite sequence $\Sigma; \Gamma; \Delta \Rrightarrow^* \Sigma^\star; \Delta^\star$, there exist a cut-free rewrite sequence $\Sigma; \Gamma; \Delta \Rrightarrow^* \Sigma^\star; \Delta^\star$.

With $\multimap_1$ replaced with $\multimap_1'$ and the cut rules shown to be redundant, the rewriting interpretation of LV is succinctly described by the rules in Figure 4. Notational conventions and structural properties are as in Figure 3. We will refer to these rules as system ω, and to their use as ω-rewriting.

2.4 Discussion

So far, we have extracted a rewriting system from a large fragment of linear logic. Before assessing the rewriting merits of ω in sections to come, we shall conclude this part with reflections on our methodology and comparisons with related ideas from the literature. We start by remarking on a few natural questions, although proper answers will be sought in future work.

First: *What about the other connectives of linear logic?* The remaining operators of minimal intuitionistic logic are $\oplus$ and its unit $\mathbf{0}$. An ω-style reading of the left rule of $\oplus$:

$$\frac{\Gamma; \Delta, A_1 \longrightarrow_\Sigma C \quad \Gamma; \Delta, A_2 \longrightarrow_\Sigma C}{\Gamma; \Delta, A_1 \oplus A_2 \longrightarrow_\Sigma C} \oplus_1$$

seems to require a form of synchronization between two possibly infinite rewrite chains. We do not understand this rule as a rewriting operation at this stage. Its nullary form, $\mathbf{0}_1$, suggests instead a reading of $\mathbf{0}$ as a "mirage" operator, as anything can be observed in its presence. Moving to a multiple conclusion sequent form in the style of FILL [9], the left rule for $\parr$:

$$\frac{\Gamma; \Delta_1, A_1 \longrightarrow_\Sigma \Theta_1 \quad \Gamma; \Delta_2, A_2 \longrightarrow_\Sigma \Theta_2}{\Gamma; \Delta_1, \Delta_2, A_1 \parr A_2 \longrightarrow_\Sigma \Theta_1, \Theta_2} \parr_1$$

seems to endow multiplicative disjunction with a rewriting semantics that splits the state and starts two completely independent computations. However, further research is required to validate this reading and extend the current work to multiple conclusion sequents. We did not venture in the realm of classical linear logic.

Interestingly, the connectives currently comprising ω coincides with the fragment of linear logic at the core of the type-theoretic logical framework for concurrency CLF [16, 47]. We do not know at this stage if there is more to this than a mere coincidence.

Second: *How sensitive is the definition of ω to the specific presentation of linear logic?* We chose LV because it elegantly capture the structural characteristics of the logic, especially as far as reusability is concerned. Its rules were amenable to a sensible rewriting interpretation, and it permitted relatively simple proofs of our various results. It is however our untested conjecture that the methodology used to derive ω can be applied to other presentations, probably with different degrees of ease. It will be interesting to compare the resulting rewrite systems.

Third: *Can this methodology be applied to other logics?* We have not tried yet, but this is a reasonable supposition. Linear logic is a good starting point because its interpretation of context formulas as consumable resources is in line with the destructive nature of rewriting. Other sub-structural logics are clearly promising candidates, but it is conceivable that interesting results could emerge from specific presentations of, say, traditional intuitionistic logic.

Fourth: *How does this compare to other proof-as-computations paradigms?* The methodology proposed here places a strong emphasis on the left rules of (linear) logic, with the right rules reduced to justifications of natural equivalences. It is worth contrasting this characteristic with the tenets of logic programming as uniform provability [36], which instead extracts the operational semantics of a logical operator from its right sequent rules. This approach has robustly been extended to linear logic programming [3, 26, 34]. In a partial departure from this short tradition, Kobayashi and Yonezawa's ACL [28] derives its semantics from specialized versions of left rules of linear logic (when examined through the lense of duality). This, together with its acceptance of open derivations and support for concurrency, makes ACL a close relative to ω. Differently from our proposal, however, it considers a limited fragment of logic, and falls short of endowing it with a rewriting interpretation. Saraswat and Lincoln hint

at a similar interpretation for their Higher-order Linear Concurrent Constraint language (HLcc) [29], interestingly stirring it in the direction of constraint programming (see also [20]). To the extent of our knowledge, ACL and HLcc are the closest proposals to ω in the literature.

Fifth: *Can logic benefit from ω?* We will see in just a few lines that ω is intimately related to various languages for concurrent computation, and can be taken to shine a logical light onto them. It remains to be investigated whether this relation can be ridden in the reverse direction as well, *i.e.*, that results and techniques from concurrency theory can find application in logic. One candidate is the very notion of derivation. We endowed ω with a semantics based on transition-*sequences*, which is common place in rewriting theory. It is however a small conceptual step to distill minimal partial orders (*traces*) by forcing sequentiality only when steps depend on each other. System ω may then carry traces over to logic, with a sound and usable notion of derivation not based on trees but on DAGs. Andreoli's "desequentialized proofs" [2] appear closely related to this idea.

3 Multiset Rewriting

Multiset rewriting captures the essence of a paradigm for concurrent and distributed computation characterized by a prominent notion of state, separate from the transitions that act upon it. Other members of this family include Petri nets [41], possibly the earliest model of concurrency, and a number of specification approaches including automata for model checking [31] and inductive definitions [40]. We will now show that various popular forms of multiset rewriting are syntactic fragments of ω-rewriting. Therefore, thanks to the logical foundations laid out in Section 2, this constitutes an interpretation of multiset rewriting *as* linear logic, which we like to contrast to the interpretations *into* linear logic traditionally found in the literature. Indeed, shortly after Girard's seminal paper on linear logic [24], Asperti noticed that it supported a simple encoding of place/transition Petri nets [4]. This observation was subsequently refined and extended by numerous authors, *e.g.*, [10, 11, 18, 25, 30].

3.1 Propositional Multiset Rewriting

We start with the most basic form of multiset rewriting, which can be seen as a notational variant of place/transition Petri nets. The language of *propositional multiset rewriting* (MSR_0 hereafter) is given by the following grammar:

Multisets	$\tilde{s}, \tilde{a}, \tilde{b}, \tilde{c} \ ::= \ \cdot \mid \tilde{s} \cdot s$
Multiset rewrite rules	$r \ ::= \ \tilde{a} \to \tilde{b}$
Rule sets	$R \ ::= \ \cdot \mid R \cdot r$

where s refers to an element of the *support set* S. Multisets $\tilde{s}$ are elements of the monoid freely generated from S, the multiset union operator "$\cdot$" and the empty multiset "$\cdot$". A rule set R is simply a set of rewrite rules.

A rule $r = \tilde{a} \to \tilde{b}$ is *applicable* in a *state* $\tilde{s}$, if $\tilde{s}$ contains r's antecedent $\tilde{a}$ (*i.e.*, $\tilde{s} = \tilde{c} \cdot \tilde{a}$). In these circumstances, the *application* of r to $\tilde{s}$ yields the state $\tilde{s}'$ obtained

by replacing $\tilde{a}$ with r's consequent $\tilde{b}$ in $\tilde{s}$ (*i.e.*, $\tilde{s}' = \tilde{c}, \tilde{b}$). This is expressed by the basic multiset rewriting judgment $\tilde{s} \triangleright_R \tilde{s}'$, which is formally defined by the following transition pattern:

$$msr_0 \quad : \qquad (\tilde{c}, \tilde{a}) \triangleright_{R, (\tilde{a} \to \tilde{b})} (\tilde{c}, \tilde{b})$$

We write $\_ \triangleright^*_R \_$ for its reflexive and transitive closure.

Propositional multiset rewriting is immediately recognized as a form of ω-rewriting by interpreting multisets as linear contexts (or tensored formulas) and rule sets as unrestricted contexts. Indeed multisets obey the same monoidal laws as contexts, and the semantic rule msr_0 can be seen as an application of rule **clone** immediately followed by $\multimap'_1$. Formally, we construct an homomorphic mapping by interpreting "$:$", "$;$", $\to$, "$\cdot$" and "$,$" as "$,$", "$\cdot$", $\multimap$, $\circ$ and $\otimes$ respectively. We naturally extend this mapping to the relative syntactic categories, and write $\ulcorner X \urcorner$ for the object in ω corresponding to entity X. The soundness of this encoding is formally stated by the following simple property:

Property 2. For every states $\tilde{s}$, $\tilde{s}'$ and every rule set R, if $\tilde{s} \triangleright^*_R \tilde{s}'$, then $S; \ulcorner R \urcorner; \ulcorner \tilde{s} \urcorner \Rrightarrow^* S; \ulcorner \tilde{s}' \urcorner$.

The family of mappings summarized as $\ulcorner \_ \urcorner$ identifies a syntactic fragment $\tilde{\omega}_0$ of ω (and linear logic). Moreover, $\ulcorner \_ \urcorner$ is a bijection over $\tilde{\omega}_0$ (modulo the monoidal laws of each formalism). It can then be shown that the inverse of the above property holds:

Property 3. For every states $\tilde{s}$, $\tilde{s}'$ and every rule set R, if $S; \ulcorner R \urcorner; \ulcorner \tilde{s} \urcorner \Rrightarrow^* S; \ulcorner \tilde{s}' \urcorner$, then $\tilde{s} \triangleright^*_R \tilde{s}'$.

Together, these properties and the trivial mapping underlying them allow us to view propositional multiset rewriting as a fragment of ω-rewriting, and therefore of linear logic. In particular, it permits redefining the semantics of MSR_0 on a purely logical basis.

3.2 First-Order Multiset Rewriting

We now extend the above results to a richer form of multiset rewriting. We consider multiset elements that can carry structured values, and are manipulated by parametric rewrite rules. Banâtre and Le Métayer have developed this basic idea into the programming language GAMMA [5], while Jensen has turned it into the flexible formalism of colored Petri nets [27]. It has recently been extended with the possibility of creating fresh data in the security specification language MSR [14]. We take this as the language of *first-order multiset rewriting* (MSR_1 hereafter).

Abstractly, we take the support set S to consist of first-order atomic formulas over some initial signature Σ_0. Rules assume the form

$$\textit{Multiset rewrite rules} \qquad r \ ::= \ \forall \boldsymbol{x}.\tilde{a} \to \exists \boldsymbol{n}.\tilde{b}$$

where $\boldsymbol{y}$ denotes a sequence of variables $(y_1, \ldots, y_n)$ for some n. The scope of the universal variables $\boldsymbol{x}$ ranges over the whole rule, while the existential variables $\boldsymbol{n}$ can appear only in its consequent. We assume implicit α-renaming for both sorts of bound variables. We write $\Sigma \vdash t$ to indicated that t is a valid term over signature Σ, and $\Sigma \vdash$

$\boldsymbol{t}$ for the natural extension of this notion to sequences of terms $\boldsymbol{t}$. We write $[\boldsymbol{t}/\boldsymbol{x}]\tilde{a}$ for the simultaneous substitution of terms $\boldsymbol{t} = (t_1, \ldots, t_n)$ for the variable $\boldsymbol{x} = x_1, \ldots, x_n$ in multiset $\tilde{a}$.

The basic judgment of MSR_1 has the form $\Sigma; \tilde{s} \triangleright_R \Sigma'; \tilde{s}'$, where both the initial and final state consist of a signature and a multiset. A rule $r = \forall \boldsymbol{x}.\tilde{a} \to \exists \boldsymbol{n}.\tilde{b}$ in R is applicable in $\Sigma; \tilde{s}$ if its universal variables can be instantiated to Σ-valid terms $\boldsymbol{t}$ so that the antecedent matches $\tilde{s}$ (*i.e.*, $\tilde{s} = \tilde{c}; [\boldsymbol{t}/\boldsymbol{x}]\tilde{a}$). In this case, applying r results in a state $\Sigma'; \tilde{s}'$ whose signature is obtained by extending Σ with $\boldsymbol{n}$ (modulo α-renaming), and $\tilde{s}'$ is given by replacing the discovered instance of $\tilde{a}$ with the corresponding instance of $\tilde{b}$ (*i.e.*, $\tilde{s}' = \tilde{c}; [\boldsymbol{t}/\boldsymbol{x}]\tilde{b}$). This is summarized by the following schematic transition:

$$msr_1\colon \Sigma; (\tilde{c}; [\boldsymbol{t}/\boldsymbol{x}]\tilde{a}) \triangleright_{R; (\forall \boldsymbol{x}.\tilde{a} \to \exists \boldsymbol{n}.\tilde{b})} (\Sigma, \boldsymbol{n}); (\tilde{c}; [\boldsymbol{t}/\boldsymbol{x}]\tilde{b}) \qquad \text{if } \Sigma \vdash \boldsymbol{t}.$$

Again, we write $\_ \triangleright^*_R \_$ for the finite iteration of $\_ \triangleright_R \_$.

The propositional embedding in Section 3.1 is easily extended to account for the first-order infrastructure just discussed: we shall simply map the rule binders $\forall$ and $\exists$ to the homonymous quantifiers $\forall$ and $\exists$ of ω. Then the semantic rule msr_1 compounds an ω-rewrite sequence consisting of rule $\mathbf{clone}$, zero or more uses of $\forall_!$, one application of $\multimap'_!$, and zero or more of $\exists_!$.

The resulting mapping, which we still call $\ulcorner \_ \urcorner$, identifies another fragment $\tilde{\omega}_1$ of ω, and is again bijective over this fragment. The formal correspondence between MSR_1 and system ω is captured by the following property [13]:

Property 4. For every signatures Σ Σ', states $\tilde{s}$, $\tilde{s}'$, and rule set R, we have that $\Sigma; \tilde{s} \triangleright^*_R \Sigma'; \tilde{s}'$ if and only if $\Sigma; \ulcorner R \urcorner; \ulcorner \tilde{s} \urcorner \Rrightarrow^* \Sigma'; \ulcorner \tilde{s}' \urcorner$.

Again, this result not only logically justifies the semantics of MSR_1, but allows viewing this language as a fragment of ω, and ultimately of linear logic.

As noted in [33], a similar relation between MSR_1 and linear logic does not hold in the reverse direction. It holds here because of the restricted use of rule $\exists_!$ embedded in ω (and the incompleteness of this system w.r.t. linear logic).

3.3 Discussion

From the above discussion, it is clear that MSR_1 accounts only for a very small fragment ($\tilde{\omega}_1$) of ω. We will now explore what else ω has to offer as a rewriting framework, and relate it to proposals in the Petri net and multiset rewriting communities.

In a major departure from traditional state-based formalisms, ω dissolves the boundary between states (usually flat collections of strictly atomic elements, even when carrying structured data) and the actuators of state change (rules). Indeed, objects of the form $A \multimap B$ can appear in the linear context, where they are responsible for the rewriting behavior in $\tilde{\omega}_1$. In this way, ω not only internalizes the rewriting operation within the state, but also makes it available for manipulation as a first-class object.

Furthermore, ω replaces the monolithic transition rules of traditional state-based languages with a toolkit of elementary state transformers drawn from the ranks of linear logic: $\otimes$ and $\mathbf{1}$ (or "," and "$\cdot$") are the basic glue, $\multimap$ expresses rewrite, ! is a reusability mark, $\forall$ introduces parameters, $\exists$ allows generating fresh data, $\&$ offers choice, and $\top$

is the unusable object. Complex transformations can easily be assembled by composing basic operator: an MSR_1 rule is an example, $(a \multimap b) \,\&\, !(c \multimap \mathbf{1})$ is another.[6]

Embedded rewrites, such as $(a \multimap b, (c, d \multimap e))$,[7] are a particularly important case of composition as they allow dynamically modifying the rule set available for rewriting. This will be our bridge to process algebra in the next section.

Similar ideas have been incorporated in enhanced forms of Petri nets, and to a lesser extent into multiset rewriting. Indeed, Valk argued for self-modifying nets as far back as 1978. A number of recent proposals, such as Hierarchical or Object Petri Nets [21, 46], fully realize this program by permitting nets to manipulate other nets, often using reflection to move between levels. Among them, Farwer and Misra's Linear Logic Petri Nets [22] are rather interesting as they operate on embedded linear logic formulas. On the multiset rewriting side, Le Métayer outlined a higher-order extension to GAMMA [32], which blurs the distinction between state and rules.

Most of these proposals are motivated by software engineering considerations, often modularity and control, sometimes inspired by process algebra. The resulting formalisms tend to be powerful but also complex, as they build on the already heavy definitions of Petri nets. It is however conceivable that they enjoy embeddings in ω akin to those sketched in Sections 3.1 and 3.2. This would endow these extensions with a formal justification in (linear) logic, and possibly enable simpler presentations.

4 A Logical Bridge to Process Algebra

Formalisms such as the π-calculus [38] support an alternative, *process-based*, representation of distributed and concurrent systems. It shuns the global state and static collection of transitions of multiset rewriting and other state-based models in favor of evolving communicating processes that tie together the data and the program of an agent, at the same time blurring the distinction between them.

We will show in this section that ω is closely related to two such process algebras: the asynchronous π-calculus [38] and the join calculus [23]. As we do so, we will focus on them as computation rather than analysis mechanisms. In particular, we will concentrate a trace-based semantics, leaving the investigation of finer notions, such as bisimulation, for future work.

4.1 The Asynchronous π-Calculus

In its minimal form, the asynchronous π-calculus (hereafter $a\pi$) considers processes defined by the following grammar [44]:

$$P, Q, R \ ::= \ \mathbf{0} \ \mid \ P \parallel Q \ \mid \ !P \ \mid \ \nu x.P \ \mid \ x(y)P \ \mid \ \overline{x}\langle y \rangle$$

where x and y are *names* (or *channels*). Hiding ($\nu x.P$) and input over a channel x ($x(y)P$) bind the names x and y respectively, up to α-renaming. We write $\mathrm{FN}(P)$ for

[6] "Either turn an a in a b once, or delete arbitrarily many c's."

[7] "Upon encountering an a, transform it into a b and introduce a single-use rule that will transform a c and a d into an e when these object appear in the state."

the set of names free in process P and $[x/y]P$ for the substitution (renaming) of x for y in P. Input and output ($\overline{x}\langle y\rangle$) are monadic, and the latter can only be the last action of a process (together with $\mathbf{0}$), which makes communication asynchronous. This core calculus can easily be generalized to support polyadic channels, complex terms, and pattern matching.

Processes are endowed with a notion of structural equivalence, written $P \overset{\pi}{\equiv} Q$, given in the following table:

$P \parallel Q \overset{\pi}{\equiv} Q \parallel P$	$\nu x.\nu y.P \overset{\pi}{\equiv} \nu y.\nu x.P$
$P \parallel \mathbf{0} \overset{\pi}{\equiv} P$	$\nu x.\mathbf{0} \overset{\pi}{\equiv} \mathbf{0}$
$P \parallel (Q \parallel R) \overset{\pi}{\equiv} (P \parallel Q) \parallel R$	$\nu x.(P \parallel Q) \overset{\pi}{\equiv} P \parallel \nu x.Q$ if $x \notin \mathrm{FN}(P)$
$!P \overset{\pi}{\equiv} P \parallel !P \quad (!)$	

It makes parallel composition ($\parallel$) a monoidal operator with the null process $\mathbf{0}$ its unit (top left). It also partially allows hiding to commute with parallel composition and other hiding operators (right). Finally, $(!)$ interprets process replication (!P) as the parallel composition of arbitrarily many copies of P (bottom left). With the exception of this last relation, we can already see a strong relation with the structural equality ($\equiv$) of ω.

Processes evolve through communication. In its basic form, it is modeled by the judgment $P \rightarrowtriangle Q$, and defined by the following inference patterns:

$$\frac{}{\overline{x}\langle y\rangle \parallel x(z)P \rightarrowtriangle [y/z]P}\ \mathbf{i/o} \qquad \frac{P \rightarrowtriangle P'}{P \parallel Q \rightarrowtriangle P' \parallel Q}\ \mathbf{cgr}\parallel \qquad \frac{P \rightarrowtriangle P'}{\nu x.P \rightarrowtriangle \nu x.P'}\ \mathbf{cgr}\nu$$

The first rule formalizes the transmission of a name y over a channel x (*reaction*). The remaining two entail that parallel composition and hiding are permeable to communication, but that replication and input block it. The structural equivalence $\overset{\pi}{\equiv}$ can implicitly massage processes before and after communication. Let $\_ \twoheadrightarrow^{*} \_$ be the reflexive and transitive closure of $\_ \rightarrowtriangle \_$.

We define an encoding $\ulcorner \_ \urcorner$ of the asynchronous π-calculus into ω as follows: first we map homomorphically $\mathbf{0}$, $\parallel$, ν and ! to $\mathbf{1}$ (or "$\cdot$"), $\otimes$ (or ","), $\exists$ and !, respectively. Then, we reserve a binary predicate symbol c and use it as a universal channel when representing input and output: $\ulcorner \overline{x}\langle y\rangle \urcorner = \mathsf{c}(x, y)$ and $\ulcorner x(y)P \urcorner = \forall y.\, \mathsf{c}(x, y) \multimap \ulcorner P \urcorner$, where $\ulcorner P \urcorner$ is the encoding of the embedded process P. Once more, $\ulcorner \_ \urcorner$ identifies a fragment $\omega_{a\pi}$ of ω so that any formula A in this fragment can unambiguously be interpreted as the representation of some process P, *i.e.*, $A = \ulcorner P \urcorner$.

The expected soundness of $\ulcorner \_ \urcorner$ over execution does not hold in general as there are reaction chains $P \twoheadrightarrow^{*} Q$ that have no counterpart in ω. A close examination of that proof attempt reveals that it is not the case that $\ulcorner P \urcorner \equiv \ulcorner Q \urcorner$ whenever $P \overset{\pi}{\equiv} Q$. The structural equivalence we labeled $(!)$ is the reason of this failure: $\Gamma; A \otimes !A \longrightarrow_{\Sigma} !A$ is not derivable in linear logic (although the reverse entailment does hold). If $(!)$ did have a counterpart, $\ulcorner \_ \urcorner$ would preserve the semantics of $a\pi$, as expressed by the following hypothetical result:

Property 5. Let P be a process and $\Sigma_P = \mathsf{c}, \mathrm{FN}(P)$.

- If $P \twoheadrightarrow^{*} Q$, then there are Σ, Γ and Δ such that $\Sigma_P; \circ; \ulcorner P \urcorner \Rightarrow^{*} (\Sigma_P, \Sigma); \Gamma; \Delta$ where $\exists \Sigma.\, !\Gamma \otimes \Delta \equiv \ulcorner Q \urcorner$ modulo $!A \equiv A \otimes !A$.

– If $\Sigma_P; \circ; \ulcorner P \urcorner \Rrightarrow^* (\Sigma_P, \Sigma); \Gamma; \Delta$ and $\exists \Sigma. !\Gamma \otimes \Delta \equiv \ulcorner Q \urcorner$, then $P \multimap^* Q$.

Since $!A \equiv A \otimes !A$ interpreted as mutual derivability does not hold in linear logic, it is clear that our encoding, or maybe linear logic itself (as the same issue is cited in [16, 33, 43]), does not accurately capture execution in the π-calculus, as traditionally defined. It has however been observed that the right-to-left reading of this equivalence is of difficult implementability, which suggest an alternative execution model in which only half of $(!)$ is kept, in the form of an added case in the definition of $\_ \multimap \_$:

$$\frac{}{!P \multimap !P \parallel P}$$

This, which corresponds exactly to rule $!_1$ in ω, turns the above property into an exact correspondence. Therefore, this amended language, that we shall call $a\pi'$, can be seen as fragment of linear logic in the same sense as MSR_1 was identified with $\tilde{\omega}_1$ in the previous section, but $a\pi$ itself cannot.

In the sequel, we will consider a language $a\pi^+$ that extends $a\pi'$ with terms over some signature Σ, polyadic channels, and pattern matching, but does never hide names used as channels. Because of this last aspect, the easy extension of $\ulcorner \_ \urcorner$ to $a\pi^+$ does not need to rely on the auxiliary symbol c. A strong version of Property 5 is valid for this language, so that it can be seen as a fragment $\omega_{a\pi^+}$ of linear logic.

4.2 The Join Calculus

The asynchronous core of the join calculus is defined by the following grammar [23]:

$$\begin{array}{rcl} P, Q, R & ::= & \mathbf{0} \mid P \parallel Q \mid \mathsf{def}\ D\ \mathsf{in}\ P \mid x\langle \boldsymbol{y} \rangle \\ D, E & ::= & J \triangleright P \mid D \wedge E \mid \top \\ J, I & ::= & J \parallel I \mid x\langle \boldsymbol{y} \rangle \end{array}$$

Processes P consist of the parallel composition of messages over polyadic channels x ($x\langle \boldsymbol{y} \rangle$) and definitions ($\mathsf{def}\ D\ \mathsf{in}\ P$). A *definition* D is a collection of *rules* ($J \triangleright P$) where each *join pattern* J is given by one or more messages patterns (also $x\langle \boldsymbol{y} \rangle$). The name tuples $\boldsymbol{y}_D$ in the pattern of a definition $D = J \triangleright P$ are bound in P, while the channel names $\boldsymbol{x}_D$ are defined. A definition $\mathsf{def}\ J_1 \triangleright P_1 \wedge \ldots \wedge J_n \triangleright P_n\ \mathsf{in}\ Q$ binds all channel names $\boldsymbol{x}_{J_i}$ in each P_j and Q. Bound names are subject to implicit α-conversion. We write $\mathrm{FN}(P)$ for the free names of a process P (and similarly definitions), and $[\boldsymbol{z}/\boldsymbol{y}]P$ for the simultaneous capture-avoiding substitution of names $\boldsymbol{z}$ for $\boldsymbol{y}$ in process P.

The join calculus defines a structural congruence, written $\equiv_j$, which specifies that processes (resp. rules) form a monoid with operation $\parallel$ (resp. $\wedge$) and unit $\mathbf{0}$ (resp. $\top$). It moreover comprises the following equivalences for definitions:

$$\boxed{\begin{array}{rcl} \mathsf{def}\ \top\ \mathsf{in}\ P & \equiv_j & P \\ (\mathsf{def}\ D\ \mathsf{in}\ P) \parallel Q & \equiv_j & \mathsf{def}\ D\ \mathsf{in}\ (P \parallel Q) \\ & & \text{if } \boldsymbol{x}_D \cap \mathrm{FN}(Q) = \emptyset \\ \mathsf{def}\ D\ \mathsf{in}\ (\mathsf{def}\ E\ \mathsf{in}\ P) & \equiv_j & \mathsf{def}\ (D \wedge E)\ \mathsf{in}\ P \\ & & \text{if } \boldsymbol{x}_E \cap \mathrm{FN}(D) = \emptyset \end{array}}$$

A process can always be $\equiv_j$-converted to the canonical form $\mathsf{def}\ D\ \mathsf{in}\ P$, where P does not contain definitions.

The operational semantics of the join calculus is expressed by the judgment $P \blacktriangleright Q$ given by the following rule, up to $\equiv_j$:

$$\begin{array}{r} \mathsf{def}\ (J \triangleright P) \wedge D\ \mathsf{in}\ ([\boldsymbol{z}/\boldsymbol{y}_J]J \parallel Q) \\ \blacktriangleright\ \mathsf{def}\ (J \triangleright P) \wedge D\ \mathsf{in}\ ([\boldsymbol{z}/\boldsymbol{y}_J]P \parallel Q) \end{array}$$

That is, whenever an instance $[\boldsymbol{z}/\boldsymbol{y}_J]J$ of the join pattern J of a rule $J \triangleright P$ appears the body of a canonical process, it can be replaced with the corresponding instance $[\boldsymbol{z}/\boldsymbol{y}_J]P$ of the rule's right-hand side P. As usual, we write $\_ \blacktriangleright^* \_$ for the reflexive and transitive closure of $\_ \blacktriangleright \_$.

We define a mapping of the various syntactic classes of the join calculus into ω. As usual, we write it $\ulcorner \_ \urcorner$, overloading this notation for processes, rules and patterns. This mapping, which is spelled out below, homomorphically maps the monoids of the join calculus to the tensorial core of ω. Similarly to the π-calculus, messages and patterns are rendered with the help of a family of auxiliary symbols $\mathbf{c}$ of increasing arity (to accommodate the names $\boldsymbol{y}$ in $x\langle \boldsymbol{y} \rangle$). We rely on ω's universal quantifier to govern the bound variables $\boldsymbol{y}_J$ of a rule $J \triangleright P$, while $\exists$ is needed to bind the variables $\boldsymbol{x}_D$ defined in a definition. The transition potential of rules is captured by means of linear implication, while their reusability is naturally expressed using !. Altogether, we have the following definition for $\ulcorner \_ \urcorner$:

$$\begin{array}{lrcl} P: & \ulcorner \mathbf{0} \urcorner & = & \cdot \\ & \ulcorner P \parallel Q \urcorner & = & \ulcorner P \urcorner, \ulcorner Q \urcorner \\ & \ulcorner \mathsf{def}\ D\ \mathsf{in}\ P \urcorner & = & \exists \boldsymbol{x}_D.\, (\ulcorner D \urcorner, \ulcorner P \urcorner) \\ & \ulcorner x\langle \boldsymbol{y} \rangle \urcorner & = & \mathsf{c}(x, \boldsymbol{y}) \\ D: & \ulcorner J \triangleright P \urcorner & = & !\forall \boldsymbol{y}_J.\, (\ulcorner J \urcorner \multimap \ulcorner P \urcorner) \\ & \ulcorner D \wedge E \urcorner & = & \ulcorner D \urcorner, \ulcorner E \urcorner \\ & \ulcorner \top \urcorner & = & \cdot \\ J: & \ulcorner J \parallel I \urcorner & = & \ulcorner J \urcorner, \ulcorner I \urcorner \\ & \ulcorner x\langle \boldsymbol{y} \rangle \urcorner & = & \mathsf{c}(x, \boldsymbol{y}) \end{array}$$

As in previous cases, this encoding is invertible, so that every formula A in its image identifies an object X of the appropriate syntactic category in the join calculus. We write ω_J for the fragment of ω in the image of $\ulcorner \_ \urcorner$.

The encoding we just defined is next shown to preserve the operational semantics of the join calculus, without any of the glitches of $a\pi$. This is formalized in the following property.

Property 6. Let P be a process and $\Sigma_P = \mathbf{c}_{\text{„}} \mathrm{FN}(P)$. Then, $P \blacktriangleright^* Q$ if and only if there are Σ, Γ, Δ and Q' such that $\Sigma_P; \circ; \ulcorner P \urcorner \Rightarrow^* (\Sigma_P \text{„} \Sigma); \Gamma; \Delta$ such that $\exists \Sigma.\, !\Gamma \otimes \Delta \equiv \ulcorner Q' \urcorner$ and $Q' \equiv_j Q$.

Once more, this result allows interpreting the language under examination as a fragment ω_J of ω, and therefore of linear logic.

4.3 Discussion

Once more, our encoding of $a\pi$ makes use of a fraction of the syntax of ω. In particular, $\top$ and $\&$ are not used at all and at most one object appears in the antecedent of $\multimap$. It is natural to interpret $\&$ as a form of non-deterministic choice. Its semantics in ω is different, however, from the choice operator, $+$, found in the synchronous π-calculus [38], as the reaction rule of the latter realizes both choice and communication in the same step. As noted in [16], its emulation with $\&$ would be sound, but in general incomplete as intermediate stages are visible in ω. Multi-antecedent implications would allow concurrent communications to occur atomically, in a fashion not dissimilar from the operational behavior of the join calculus [23].

Several authors have taken to the task of giving logical interpretations to process algebras, with particular focus on the π-calculus. Operationally sound and complete CLF encodings of both the synchronous and asynchronous versions of this language are given in [16]. A propositional fragment of the π-calculus is instead analyzed in [33]. That paper attempts a logical account of a form of testing equivalence. The adaptation to ω of classical notions of inter-process equivalence goes beyond the scope of the present work, but would be particularly interesting to undertake as future work.

5 Specifying Security Protocol

With the recent surge of interest in security protocols, numerous languages have been adapted or invented for the purpose of specifying and reasoning about these subtle distributed algorithms. With a few exceptions, these languages tend to be either process-oriented or state-based. The former include the spi-calculus [1], a security-enhanced version of the π-calculus, strand spaces [19], and others such as [17]. The latter comprises formalisms directly based on multiset rewriting [12, 14], tool-specific languages [31], inductive definitions [40], and more.

This profusion of formalisms has triggered an intense research activity intent to comparing and bridging them [8, 13, 15, 17]. In spite of clear commonalities, these mappings are very specific to the languages they consider, and therefore somewhat ad-hoc and hardly reusable. With a foot in both the state- and process-based camp and easy embeddings, we propose a security-conscious version of ω as a reusable and logically motivated intermediate language for carrying on these investigations. This language, that we call MSR 3, is itself a promising formalism for the specification of cryptographic protocols, precisely because it supports both representation paradigms, and can combine them when convenient.

5.1 A Preview of MSR 3

In order to represent security protocols, we consider an initial signature Σ_s that makes available function symbols $\{\_\}\_$ and $[\_, \_]$ to symbolically express encryption and concatenation (for succinctness, we will often omit the brackets in the latter). Other cryptographic operations can be included as needed. We also require Σ_s to provide predicate symbols $\mathsf{N}(\_)$ and $\mathsf{I}(\_)$ to represent network messages in transit and intruder knowledge.

Other predicates, for example to hold values local to a principal, can also appear in Σ_s. Different forms of data can be distinguished through typing, although we will refrain from doing so here for the sake of brevity.

The language MSR 1 [14] adopts such a signature in a first-order multiset rewriting framework of the sort analyzed in Section 3.2. It is therefore a fragment of ω. In this section, we will use ω itself as a language for specifying protocols.

As often done, we will use the Needham-Schroeder public-key protocol [39] as an example. This protocol, informally described below, has the purpose of establishing communication between an initiator A and a responder B, and authenticating A to B.

$$\begin{array}{l} A \to B : \{A, n_A\}_{k_B} \\ B \to A : \{n_A, n_B\}_{k_A} \\ A \to B : \{n_B\}_{k_B} \end{array}$$

Here, A creates a fresh value (nonce) n_A and sends it together with her name to B, encrypted with B's public key. Upon successfully decrypting this message, B creates his own nonce n_B and sends it to A together with n_A. Upon recognizing n_A as her original nonce, A sends n_B back to B as an acknowledgment.

We will now express the initiator's part of this protocol in ω. We are immediately faced with the choice of which representation paradigm to use. We give both a state-based and a process-based specification. For the sake of brevity, we do not explicitly represent administrative tasks such as a principal accessing his or his interlocutor's keys (see [14]): this will allow us to concentrate on the overall structure of the specification rather than these details.

The state-based representation of the initiator role of this protocol is expressed by the following two rules:

$$\boxed{\begin{array}{l} \forall A.\, \forall k_B. \\ \quad \mathbf{1} \multimap \exists n_A.\, \mathsf{N}(\{A, n_A\}_{k_B}), \mathsf{L}(A, n_A, k_B) \end{array}}$$

$$\boxed{\begin{array}{l} \forall A.\, \forall k_B.\, \forall k_A.\, \forall n_A.\, \forall n_B. \\ \quad \mathsf{N}(\{n_A, n_B\}_{k_A}), \mathsf{L}(A, n_A, k_B) \multimap \mathsf{N}(\{n_B\}_{k_B}) \end{array}}$$

The first captures the initial step of the protocol, while the second expresses the rest. In order to ensure that these rules are executed in the proper order, they rely on the auxiliary predicate L, which has also the task of communicating the parameters of the execution to the second rule (in particular the value of n_A). This encoding resembles very closely the specification of this protocol in MSR 1 [14] and other state-based formalisms.

The process-based representation of this role does away with the auxiliary predicate L altogether in favor of a nested implication:

$$\boxed{\begin{array}{l} \forall A.\, \forall k_B. \\ \quad \mathbf{1} \multimap \exists n_A.\, \mathsf{N}(\{A, n_A\}_{k_B}), \\ \qquad\qquad \boxed{\begin{array}{l} \forall n_B.\, \forall k_A. \\ \quad \mathsf{N}(\{n_A, n_B\}_{k_A}) \multimap \mathsf{N}(\{n_B\}_{k_B}) \end{array}} \end{array}}$$

This closely resembles the description of this role in a process-based language such as strand spaces [19] or the spi-calculus [1]. Observe the nested vs. cascaded nature of the

specification. Miller has shown that, given some constraints on L, these two specifications are logically equivalent [35] (although not in the sense of $\equiv$).

Differently from all other protocol specification languages we are aware of, ω makes both styles available when expressing a protocol. Not only can the specifier chose which one is most appropriate to the task at hand, but she can mix and match them at her leisure. Indeed, the initiator and receiver roles are not even required to use the same paradigm, so that if our first specification is used, the receiver could seamlessly be process-based. This may be useful, for example, when analyzing client-server protocols for denial-of-service vulnerabilities where one may want to use the more succinct process-oriented form for the client, but a state-based representation for the server in order to clearly account for how much data is stored (in the auxiliary predicate L) between exchanges. A mixed representation may also be beneficial when representing the intruder capabilites, as a process-based encoding tends to over-sequentialize the specification [8]. We expect these benefits to grow with the size and complexity of the protocol at hand.

MSR 1 has been extended with a powerful type system into MSR 2 [12]. We similarly define the language MSR 3 as the corresponding strongly typed version of ω. A precise description of MSR 3 goes beyond the scope of this paper.

5.2 Discussion

The coexistence of both the state- and process-based paradigm in ω makes it a useful melting pot, not just as a specification tool, but also as an intermediate language when comparing different formalisms. Indeed, it is well known that the terrain between the two paradigms is bumpy and treacherous [7, 8, 15], and any new road shall reckon with these difficulties. System ω suggests a different approach: engineer a robust translation between the state- and the process-based fragments of this language, and use it as a fast expressway to relate them. Other languages can then be mapped to the closest fragment of ω by what would be neighborhood roads in our analogy.

To be more precise, we define an execution preserving embedding of all of ω in $\tilde{\omega}_1$, which we identified as the counterpart of first-order multiset rewriting, a quintessential state-based language. This encoding is rather simple and well-behaved. Space limitation prevent us from presenting it, and we shall refer the interested reader to [8], which gives a similar translation.

What fragment of ω (or what process algebra) best captures the process-based paradigm is open to discussion. Were we to take $\omega_{a\pi^+}$, we would similarly map ω to this sublanguage. See again [8] for details. This direction is not as easy, and it is not clear whether a fully satisfactory solution exists.

Now, in order to relate, say, strand spaces [19] and Paulson inductive encoding [40], it suffices to produce a shallow encoding of the former into $\omega_{a\pi^+}$, a similarly simple translation of the latter to $\tilde{\omega}_1$, and then use the two internal translations we just sketched bridge them.

6 Conclusions and Future Work

We have endowed a large fragment of linear logic with a rewriting semantics by interpreting the left sequent rules of linear logic as rewrite transitions, folding selected right rules into a structural equivalence, and extending our focus beyond finite derivations. The resulting language, which we called system ω, has been shown to embed popular forms of multiset rewriting and Petri nets, giving a clean logical reading to their semantics. We have also demonstrated ω's strong ties to process algebra, with simple execution-preserving embeddings of the join calculus and a computational variant of asynchronous π-calculus. We suggested relying on ω's position as a logical meeting point of multiset rewriting and process algebra for the purpose of expressing and reasoning about cryptographic protocols, a application area where both types of formalisms have been used, often in complementary ways.

As implied in the "Discussion" paragraphs concluding each of the above sections, this work can be extended in numerous directions. In particular, we expect the definition of ω to evolve as questions about is logical foundations are answered (see Section 2.4). Pursuing the relation with process algebraic languages is particularly interesting in light of the results in Section 4 and the application potential of ω in the sphere of security protocol specification (Section 5).

Acknowledgments

We are grateful to Frank Pfenning, Dale Miller, Mark-Oliver Stehr, Valeria de Paiva, Gerald Allwein, Steve Zdancevic and Vijay Saraswat for their comments on various aspects of this work.

References

1. M. Abadi and A. Gordon. A calculus for cryptographic protocols: the spi calculus. *Information and Computation*, 148(1):1–70, 1999.
2. J.-M. Andreoli. Focussing proof-net construction as a middleware paradigm. In A. Voronkov, editor, *Proc.CADE–18*, pages 501–516, Copenhagen, Denmark, 2002. Springer Verlag LNAI 2392.
3. J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.
4. A. Asperti. A logic for concurrency. Technical report, Computer Science Department, University of Pisa, 1987.
5. J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98–111, 1993.
6. A. Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, Laboratory for Foundations of Computer Sciences, University of Edinburgh, 1996.
7. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.
8. S. Bistarelli, I. Cervesato, G. Lenzini, and F. Martinelli. Relating Process Algebras and Multiset Rewriting for Security Protocol Analysis. In R. Gorrieri, editor, *Third Workshop on Issues in the Theory of Security — WITS'03*, pages 21–31, Warsaw, Poland, 2003.

9. T. Braüner and V. de Paiva. Cut-elimination for full intuitionistic linear logic. Technical Report RS-96-10, BRICS, Denmark, 1996.
10. C. Brown and D. Gurr. A categorical linear framework for Petri nets. In *Proc. LICS'90*, pages 208–218, Philadelphia, PA, 1990. IEEE Computer Society Press.
11. I. Cervesato. Petri nets and linear logic: a case study for logic programming. In M. Alpuente and M. I. Sessa, editors, *Proc. GULP–PRODE'95*, pages 313–318, Marina di Vietri, Italy, 1995.
12. I. Cervesato. Typed MSR: Syntax and examples. In *1st International Workshop on Mathematical Methods, Models and Architectures for Computer Networks Security — MMM'01*, pages 159–177. Springer-Verlag LNCS 2052, 2001.
13. I. Cervesato, N. Durgin, M. Kanovich, and A. Scedrov. Interpreting strands in linear logic. In H. Veith, N. Heintze, and E. Clark, editors, *2000 Workshop on Formal Methods and Computer Security*, Chicago, IL, 2000.
14. I. Cervesato, N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *Proc. CSFW'99*, pages 55–69, Mordano, Italy, 1999. IEEE Computer Society Press.
15. I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Relating strands and multiset rewriting for security protocol analysis. In *13th IEEE Computer Security Foundations Workshop — CSFW'00*, pages 35–51, Cambrige, UK, 2000. IEEE Computer Society Press.
16. I. Cervesato, F. Pfenning, D. Walker, and K. Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Computer Science Department, Carnegie Mellon University, 2002.
17. F. Crazzolara and G. Winskel. Events in security protocols. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 96–105, Philadelphia, PA, 2001. ACM Press.
18. U. Engberg and G. Winskel. Petri nets as models of linear logic. In A. Arnold, editor, *15th Colloquium on Trees in Algebra and Programming*, pages 147–161, Copenhagen, Denmark, 1990. Springer-Verlag LNCS 431.
19. T. Fábrega, J. Herzog, and J. Guttman. Strand spaces: Why is a security protocol correct? In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 160–171, Oakland, CA, May 1998. IEEE Computer Society Press.
20. F. Fages, P. Ruet, and S. Soliman. Phase semantics and verification of concurrent constraint programs. In *Proceedings of the 13th IEEE Colloquium on Logic in Computer Science — LICS'98*.
21. B. Farwer. A linear logic view of object Petri nets. *Fundamenta Informaticae*, 37(3):225–246, 1999.
22. B. Farwer and K. Misra. Dynamic modification of system structures using LLPNs. In *Proc. 5th Conference on Perspectives of System Informatics*, pages 177–190, Novosibirsk, Russia, 2003.
23. C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proc. POPL'96*, pages 372–385. ACM Press, 1996.
24. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
25. C. Gunter and V. Gehlot. Nets as tensor theories. In *10th International Conference on Application and Theory of Petri Nets*, pages 174–191, Bonn, Germany, 1989.
26. J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.
27. K. Jensen. Coloured Petri nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets*, pages 248–299. Springer-Verlag LNCS 254, 1986.
28. N. Kobayashi and A. Yonezawa. ACL — A concurrent linear logic programming paradigm. In D. Miller, editor, *Proc. ILPS'03*, pages 279–294, Vancouver, Canada, 1993. MIT Press.

29. P. Lincoln and V. Saraswat. Higher-order, linear, concurrent constraint programming. Manuscript, 1993.
30. N. Martí-Oliet and J. Meseguer. From Petri nets to linear logic. *Mathematical Structures in Computer Science*, 1:66–101, 1991. Revised version of paper in LNCS 389.
31. C. Meadows. The NRL protocol analyzer: an overview. In *2nd International Conference on the Practical Applications of Prolog*, 1994.
32. D. L. Métayer. Higher-order multiset programming. In *Proc. DIMACS workshop on specifications of parallel algorithms*. American Mathematical Society, DIMACS series in Discrete Mathematics, Vol. 18, 1994.
33. D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proc. ELP*, pages 242–265. Springer-Verlag LNCS 660, 1992.
34. D. Miller. A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996.
35. D. Miller. Encryption as an abstract data-type: An extended abstract. In I. Cervesato, editor, *Proc. FCS*, pages 3–14, Ottawa, Canada, 2003.
36. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
37. D. Miller and A. Tiu. A proof theory for generic judgments: An extended abstract. In *Proceedings of LICS 2003*, pages 118–127. IEEE, June 2003. Extended version available at `http://www.cse.psu.edu/~dale/papers/nabla-subm.pdf`.
38. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.
39. R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
40. L. C. Paulson. Proving properties of security protocols by induction. In *Proc. CSFW'97*. IEEE Computer Society Press, 1997.
41. C. A. Petri. Fundamentals of a theory of asynchronous information flow. In *Proc. IFIP*, pages 386–390, Amsterdam, 1963. North Holland Publ. Comp.
42. F. Pfenning. Structural cut elimination. In D. Kozen, editor, *Proc. LICS'95*, pages 156–166, San Diego, CA, 1995. IEEE Computer Society Press.
43. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.
44. D. Sangiorgi and D. Walker. *The π-Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
45. R. A. G. Seely. Linear logic, $\star$-autonomous categories and cofree coalgebras. In J. W. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic*, pages 371–382. American Mathematical Society, 1989. Proceedings of the AMS-IMS-SIAM Joint Summer Research Conference, June 14–20, 1987, Boulder, Colorado; Contemporary Mathematics Volume 92.
46. R. Valk. Petri nets as token objects: An introduction to elementary object nets. In J. Desel and M. Silva, editors, *19th International Conference on Application and Theory of Petri Nets*, pages 1–25, Lisbon, Portugal, 1998. Springer-Verlag LNCS 1420.
47. K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Computer Science Department., Carnegie Mellon University, 2002.
48. J. Zucker. Formalisation of classical mathematics in AUTOMATH. In *Colloques Internationaux du Centre National de Recherche Scientifique*, volume 249, pages 135–145, July 1975.