Choreographic Compilation of Decentralized Comprehension Patterns*

Iliano Cervesato, Edmund S. L. Lam, and Ali Elgazar

Carnegie Mellon University
iliano@cmu.edu, sllam@andrew.cmu.edu and aee@cmu.edu

Abstract. We develop an approach to compiling high-level specifications of distributed applications into code that is executable on individual computing nodes. The high-level language is a form of multiset rewriting augmented with comprehension patterns. It enables a programmer to describe the behavior of a distributed system as a whole rather than from the perspective of the individual nodes, thus dramatically reducing opportunities for programmer errors. It abstracts away the mechanics of communication and synchronization, resulting in concise and declarative specifications. Compilation generates low-level code in a syntactic fragment of this same formalism. This code forces the point of view of each node, and standard state-of-the-art execution techniques are applicable. It is relatively simple to show the correctness of this compilation scheme.

1 Introduction

Rule-based programming, a model of computation by which rules modify a global state by concurrently rewriting disjoint portions of it, is emerging as an effective paradigm for implementing complex distributed applications [1,4,8,12]. Rule-based languages are declarative, which promises simpler reasoning than conventional languages, and even a safeguard against many of the pitfalls of concurrency [2]. Their main benefit, however, is that they can capture the behavior of a distributed application as a single entity [8], giving the programmer a bird's-eye view that abstracts away the tedium of explicitly managing communication and the intricacies of implementing synchronization. The resulting *system-centric specifications* are concise, high-level, and again declarative. Now, because a distributed application ultimately runs on an ensemble of communicating devices, such system-centric specifications need to be compiled into code that runs on the individual devices, *node-centric code*. The translation from high-level system-centric specifications to lower level node-centric code is called *choreographic compilation* [9]. It automatically weaves in the code that handles messaging and synchronization, which are notorious sources of concurrency bugs (especially in the hands of novice programmers). Choreographic compilation is especially effective when the resulting code is in a fragment of the source, rule-based, language, as their declarative nature enables simple proofs of correctness, verifiable complexity bounds, and other forms of assurance.

* This paper was made possible by grants NPRP 4-341-1-059, NPRP 4-1593-1-260, and JSREP 4-003-2-001, from the Qatar National Research Fund (a member of the Qatar Foundation). The statements made herein are solely the responsibility of the authors.

In this paper, we develop a choreographic compilation scheme for a specific class of rule-based languages, namely multiset rewriting languages with support for multiset comprehension patterns. *Multiset rewriting languages* represent the state of an ensemble as a multiset of located facts, each describing information held by a participating node. Computation happens by applying rules that rewrite a fixed number of facts into new facts. *Comprehension patterns* allow a programmer to write rules that operate not only on a fixed multiset of facts, but on all the facts that match a given pattern, ensemble-wide. This yields more readable, concise and declarative programs that coordinate large amounts of data or use aggregate operations. We implemented this idea into Comingle [8], a rule-based language for programming mobile distributed applications.

Compiling comprehension patterns in a distributed setting requires addressing the compounded effects of two challenges. The first is that multiset rewriting rules are executed *atomically*. This entails that a high-level rule, which may involve multiple locations, needs to be compiled into a set of node-centric rules, each taking the point of view of a single location, plus coordination rules that provide the illusion of atomicity [9]. The second challenge is that comprehension patterns operate *maximally* [6]: they identify *all* facts that match them in the ensemble.

We limit the discussion to the common rule format where one node has a direct connection to all other nodes participating in the rule, thereby ruling out multi-hop communications. At its core, atomicity is achieved by running a two-phase commit protocol centered on this primary node. A naive way to achieve maximality is to lock all nodes involved, so that no concurrently executing code can consume or add facts while this rule is undergoing piecemeal execution. We mitigate the obvious adverse effect on performance by locking only facts that appear in relevant comprehension patterns. This already gives all Comingle programs we have developed an acceptable running time.

Altogether, this paper makes the following main contributions:

- We identify a practical class of system-centric rules with comprehensions that enable effective choreographic compilation.
- We give a mathematical description of this transformation for a large fragment.
- We prove that the node-centric code produced by this compilation scheme retains the behavior of the source system-centric program.

Section 2 of this paper introduces our language through an example, with Section 3 formally defining it. We discuss rule topology in Section 4 and give selected details of our choreographic compilation scheme in Section 5. Correctness results are presented in Section 6. We review related work in Section 7 and outline further developments in Section 8. Omitted details can be found in a companion technical report [7].

2 A Motivating Example

Consider the problem of computing the average temperature from the readings of an ensemble of networked sensors. Traditionally, this involves writing at least three programs: one that probes each sensor, computes the average temperature and reports the result; the second is a sensor-side program that returns a reading when probed; the last expects the result.

Comingle takes a different approach. The information held by each device is stored as a series of *facts*. For example, a temperature reading of 16.3 degrees could be expressed as the fact $temp(16.3)$. We visualize the node where a fact is held as a *located fact*, writing for example $[\ell_{23}]temp(16.3)$ to express that the reading at node ℓ_{23} is 16.3 degrees. Located facts are used for all kinds of information. Here, the topology of the sensor network could be given as located facts of the form $[\ell]neighbor(\ell')$, expressing that ℓ' is directly connected to ℓ. Similarly, the request for node ℓ to compute the temperature average A of its neighbors and report it to node ℓ' would be written as located facts $[\ell]getAvg(\ell')$ and $[\ell']report(A)$, respectively.

Programs in Comingle take the form of a collection of rules that consume some of the facts held in the ensemble and replace them with other facts, possibly at different nodes. For our example, a single rule suffices. A buggy solution that always reports 25.0 degrees without consulting the sensors would have the form

$$\forall X, Y.\, [X]getAvg(Y) \multimap [Y]report(25.0) \tag{1}$$

This rule is parametric in the locations involved: X is the node computing the average and Y is the location where to deliver the result. Whenever the ensemble contains an instance, say $[\ell_{12}]getAvg(\ell_9)$, of the left-hand side, this rule can be applied with the effect of replacing this fact with $[\ell_9]report(25.0)$. Observe that this effect is global: it consumes a fact from one node and creates a related fact in a different node. This reading makes rule (1) *system-centric* as it describes a computation that views the ensemble as a single entity.

But of course this rule comes short of correctly solving our problem. Node X, which does the polling, needs to collect the temperature of all its neighbors. Comingle provides *multiset comprehension patterns* as a convenient primitive for this kind of actions. The comprehension pattern $\wr[X]neighbor(N)\int_{N\to Ns}$ collects all the neighbors N of X into a multiset Ns. While this is a local computation occurring at X, comprehension patterns do not need to be local: the comprehension $\wr[N]temp(T) \mid N \in Ns\int_{T\to Ts}$ collects the temperature reading $[N]temp(T)$ held at each node N among Ns into a multiset Ts. At this point, the average is simply computed by adding up the values in Ts and dividing by the number of such values, all primitive operations in Comingle. The overall computation is captured by the rule

$$\forall \left[\begin{array}{l} \wr[X]neighbor(N)\int_{N\to Ns}, \\ \wr[N]temp(T) \mid N \in Ns\int_{T\to Ts} \end{array}\right] \setminus\ [X]getAvg(Y) \multimap [Y]report(A) \quad \text{where } A = sum(Ts)/size(Ts) \tag{2}$$

where the facts matched by the expressions before "\" are consulted but not deleted by the rule application, while the fact after it is consumed. The "where" clause denotes a side computation, something we will generalize into the notion of a guard. Both are convenience syntax that are not part of the core language.

Rule (2) is system-centric too, even more so than our first example. Its application is atomic and maximal: from the point of view of the programmer, the matching of facts on the left-hand side of $\multimap$ and the rewriting on its right-hand side happen in one go, moreover *all* facts matching $[X]neighbor(N)$ are collected in Ns, and similarly for Ts.

Variables: x Locations: ℓ Terms: t Guards: g Predicates: p

Base Facts	$f ::= p(\vec{t})$	Located Facts	$F ::= [\ell]f$
Expressions	$E ::= F \mid \wr F \mid g \int_{\vec{x} \rightleftarrows t}$	Rules	$R ::= \forall \vec{x}.\, H \mid g \multimap B$
Heads,Bodies	$H, B ::= \overline{E}$	Programs	$\mathcal{P} ::= \overline{R}$

Fig. 1. Abstract Syntax of Core Comingle

While rule (2) captures exactly the process of solving our example problem, and in a most concise way, it is impossibly abstract from the point of view of the nodes in a distributed system: such nodes are only able to send and receive messages, and perform local computation. We bridge this abstraction gap by transforming rule (2) into a set of rules that look much more like rule (1). This rule has, in fact, a simple operational interpretation in a decentralized ensemble of computing nodes: its left-hand side, $[X]getAvg(Y)$, performs some local computation at node X (here retrieving the value of a stored fact), while its right-hand side can be understood as sending the message $report(25.0)$ to node Y — the underlying networking middleware will take care of delivering it to Y as a fact that it can then use. Rule (1) has therefore also a *node-centric* interpretation, that can be used operationally. The main challenges of designing a choreographic compilation scheme for Comingle — i.e., a transformation of each abstract, system-centric, rule into an equivalent set of operational, node-centric, rules — is to maintain the illusion of atomicity and maximality at the operational level. This is the subject of the remainder of this paper and of the technical report [7].

3 Core Comingle

In this section, we formalize the core syntax and semantics of Comingle — the full language is described in [7,8]. We begin by introducing some notation. We write $\overline{o}$ for a multiset of syntactic objects o. We denote the extension of a multiset $\overline{o}$ with an object o as "$\overline{o}, o$", with $\varnothing$ indicating the empty multiset. We also write "$\overline{o}_1, \overline{o}_2$" for the union of multisets $\overline{o}_1$ and $\overline{o}_2$. The literal multiset containing $o_1, \ldots, o_n$ is denoted $\wr o_1, \ldots, o_n \int$. Given a multiset of labels $\mathcal{I}$, the multiset of objects o_i for $i \in \mathcal{I}$ is denoted $\wr \bigcup_{i \in \mathcal{I}} o_i$. We write $\vec{o}$ for a tuple of o's and $[\vec{t}/\vec{x}]o$ for the simultaneous substitution within object o of all free occurrences of variable x_i in $\vec{x}$ with the corresponding term t_i in $\vec{t}$. A generic substitution is denoted θ. Substitution implicitly α-renames bound variables as needed to avoid capture. We write $FV(o)$ for the set of free variables in o.

Syntax. Figure 1 defines the abstract syntax of Comingle. *Locations* ℓ are names that uniquely identify computing nodes, and the set $\mathcal{L}$ of all nodes participating in a Comingle computation is called an *ensemble*. At the Comingle level, computation happens by rewriting *located facts* F of the form $[\ell]p(\vec{t})$ where p is a predicate symbol and $\vec{t}$ is a tuple of *terms*. We will simply refer to them as facts. The semantics of Comingle is largely agnostic to the specific language of terms — in this paper, we assume a first-order term language extended with primitive multisets. We write $[\ell]f$ for a generic fact f located at node ℓ.

Comingle transitions: $\mathcal{P} \rhd St \longmapsto St'$

$$\frac{\forall\,(H \mid g \multimap B) \in \mathcal{P} \quad \models \theta g \quad \theta H \triangleq_{\texttt{head}} St_H \quad \theta H \triangleq^{\neg}_{\texttt{head}} St \quad \theta B \ggg_{\texttt{body}} St_B}{\mathcal{P} \rhd St_H, St \longmapsto St_B, St}\ (\textsf{rw})$$

where $H \triangleq_{\texttt{head}} St$ *iff* store St matches ground head H
$\models g$ *iff* ground guard g is satisfiable
$H \triangleq^{\neg}_{\texttt{head}} St$ *iff* store St matches no comprehension patterns in ground head H
$B \ggg_{\texttt{body}} St$ *iff* ground body B unfolds to store St

Fig. 2. Abstract Semantics of Comingle

Computation in Comingle happens by applying *rules* of the form $\forall \vec{x}.\, H \mid g \multimap B$. We refer to H as the *head* of the rule, to g as its *guard* and to B as its *body*. The head of a rule consists of *atoms* F and of *comprehension patterns* of the form $\wr F \mid g \int_{\vec{x} \to ts}$ (written $\wr F \mid g \int_{\vec{x} \leftarrow ts}$ in the body — the direction of the arrow is suggestive of the flow of information). An atom F is a located fact $[\ell]p(\vec{t})$ that may contain variables in the terms $\vec{t}$ or even as the location ℓ. Guards in rules and comprehensions are Boolean-valued expressions constructed from terms and are used to constrain the values that the variables can assume. Just like for terms we keep guards abstract, writing $\models g$ to express that ground guard g is satisfiable. Two types of guards used pervasively in this paper are term equality $t = t'$ and multiset membership $t \in ts$. We drop the guard from rules and comprehensions when it is the always-satisfiable constant $\top$. A comprehension pattern $\wr F \mid g \int_{\vec{x} \rightleftarrows ts}$ represents a multiset of facts that match the atom F and satisfy guard g under the bindings of variables $\vec{x}$ that range over ts, a multiset of tuples called the *comprehension range*. We call F the *subject* of the comprehension. The scope of $\vec{x}$ is the atom F and the guard g. We implicitly α-rename bound variables to avoid capture. A comprehension pattern $\wr [x]p(\vec{t}) \mid g \int_{\vec{x} \rightleftarrows ts}$ is *system-centric* whenever x appears in $\vec{x}$. The body B of a rule is also a multiset of atoms and comprehension patterns.

The universal variables $\vec{x}$ in a rule $\forall \vec{x}.\, H \mid g \multimap B$ account for all the free variables in H, g and B, and we often write $\forall\,(H \mid g \multimap B)$ for succinctness. Moreover, we only consider *safe* rules where $FV(B) \subseteq FV(H, g)$. We will occasionally use rules of the form $\forall \vec{x}.\, H_r \backslash H_c \mid g \multimap B$, viewed as an abbreviation for $\forall \vec{x}.\, (H_c, H_r) \mid g \multimap (B, H_r)$; we then refer to H_r and H_c as the retained and consumed heads of the rule.

A Comingle *program* is a collection of rules.

Semantics. We describe the computation of a Comingle system by means of a small-step transition semantics. Its basic judgment has the form $\mathcal{P} \rhd St \longmapsto St'$ where $\mathcal{P}$ is a program, St is a store and St' is a store that can be reached in one (abstract) step of computation. A *store* St is a multiset of ground located facts $[\ell]p(\vec{t})$.

Rule (rw) in Figure 2 describes a step of computation that applies a rule $\forall\,(H \mid g \multimap B)$. This involves identifying a closed instance of the rule obtained by means of a substitution θ. The instantiated guard must be satisfiable ($\models \theta g$) and we must be able to partition the store into two parts St_H and St. The instance of the head must match St_H ($\theta H \triangleq_{\texttt{head}} St_H$), while the remaining fragment St must not match any comprehension in it ($\theta H \triangleq^{\neg}_{\texttt{head}} St$). The rule body instance θB is then unfolded ($\theta B \ggg_{\texttt{body}} St_B$) into

St_B which replaces St_H in the store. A reading of these auxiliary judgments is given in Figure 2. A formal description can be found in [7,8].

Rule (rw) embodies a system-centric abstraction of the rewriting semantics of Comingle as it atomically accesses facts at arbitrary locations. Indeed, it views the facts of all participating locations in the ensemble as one virtual collection. This abstract notion of rule application needs to be compiled into a concurrent, node-centric model of computation, where each node manipulates its local facts and sends messages to other nodes.

4 Neighbor Restriction

In this section, we identify a syntactic class of Comingle rules that support efficient node-level execution. Characteristic of these *1-neighbor restricted rules* is that, in any instance, there is one node that has every other location participating in the rule as a *neighbor*. Operationally, the execution of the rule can use this *primary location* as a communication hub to all the other participating nodes, called *forwarding locations*. For brevity, we provide only the intuition behind most definitions. See [7] for full details.

To start with, consider a rule $R = \forall(H \multimap B)$ with an empty guard and without comprehension patterns in its head. A node X has Y as a its *neighbor* in R if the head H contains a fact $[X]p(\vec{t})$ such that Y occurs in $\vec{t}$. For simplicity, we take this as a proxy for a direct communication link — in actuality only certain facts may be used to describe point-to-point messaging.

Guards somewhat complicate this definition as they are often used to calculate new values, including locations, on the basis of existing values. Let g be a guard with free variables $\vec{x}$ and $\vec{y}$. We say that $\vec{x}$ *determines* $\vec{y}$ in g, written $\vec{x} \overset{g}{\Longmapsto} \vec{y}$, if for every ground substitution $\vec{t}/\vec{x}$ there is at most one substitution $\vec{s}/\vec{y}$ that makes g satisfiable, i.e., such that $\models [\vec{t}/\vec{x}, \vec{s}/\vec{y}]g$. We write $\vec{x} \overset{g}{\Longmapsto} y$ if y is among such $\vec{y}$. Then, Y is a neighbor of X in rule $\forall(H \mid g \multimap B)$ if the set of variables occurring in facts located at X in H determines Y. In symbols, $\{x \in FV(E) : E = [X]f \text{ in } H\} \overset{g}{\Longmapsto} Y$.

Comprehension patterns further complicate this definition as they may identify participating locations indirectly through their comprehension range — for example N in $\wr[N]temp(T) \mid N \in Ns\int_{T \to Ts}$ but also Ns in $\wr[X]neighbor(N)\int_{N \to Ns}$. Thus, a (possibly bound) variable Y in $\wr[Y]f \mid g_Y\int_{\vec{y} \to ts}$ is a neighbor of X in rule $\forall(H \mid g \multimap B)$ if $\{x \in FV(E) : E = [X]f' \text{ or } E = \wr[X]f' \mid g'\int_{\vec{x} \to ts'} \text{ in } H\} \overset{g, g_Y}{\Longmapsto} Y$.

Given this definition of neighbor, a location X_n is n *hops away* from X_0 in rule R if n is the smallest number such that there are nodes $X_1, \ldots X_{n-1}$ such that X_i has X_{i+1} as its neighbor for each i from 0 to $n-1$. Rule R is n*-neighbor restricted* with *primary location* X, something we denote $\vdash^n_{\text{NB}} R \gg X$, if every location Y such that $[Y]f$ appears in R is at most n hops away from X. Each such Y other than X is called a *forwarding location*. A Comingle rule that is not n-neighbor for any n has mutually unreachable nodes and therefore cannot be concretely executed on a distributed collection of nodes as it would require out-of-band synchronization that bypasses the underlying communication infrastructure. We are particularly interested in rules where $n = 1$. In fact, 1-neighbor restricted rules are such that the primary location has a direct communication link to every other location participating in the rule, which entails that device-level code that implements it only needs to use point-to-point messaging

primitives to and from the primary location, thereby avoiding complex routing. Furthermore, 1-neighbor restricted rules where all head facts are at the primary location are such that local computation is sufficient to determine applicability, i.e., if there is a match for their head in the computing state — their body may however locate facts at other nodes. We call such rules *node-centric*. Rule (1) from Section 2 is node centric with primary location X as its head contains a single atom located at X. Rule (2) is 1-neighbor restricted with primary location X (but not node-centric) as the comprehension $\wr[X]neighbor(N)\int_{N\to Ns}$ (locally) determines the contents of the multiset Ns from which the value of every location N in $\wr[N]temp(T) \mid N \in Ns\int_{T\to Ts}$ is drawn. Thus each value T held in $[N]temp(T)$ can be accessed in one hop from X.

A Comingle program is 1-neighbor restricted if all its constituent rules are such. All applications we have developed using Comingle have naturally been 1-neighbor restricted [8], and therefore we will limit our discussion to this class of programs. We will use programs consisting solely of node-centric rules (node-centric programs) as the target of the compilation of 1-neighbor restricted programs. See [9] for a generalization in the absence of comprehensions.

5 Choreographic Transformation

Choreographic compilation elaborates each system-centric rewrite rule R into a set $[\![R]\!]$ of node-centric rewrite rules that execute portions of R at the participating locations. The challenge is to design $[\![R]\!]$ so that it behaves exactly like R, i.e., that it is applicable whenever R is and eventually achieves its effects (completeness), and that it does not introduce any new effects (soundness), especially partial execution. In Section 6, we spell out these requirements and outline proofs that our compilation satisfies them.

In the absence of comprehension patterns, Comingle is *monotonic*:

Property 1 (Monotonicity).

$$\text{If } \mathcal{P} \rhd St \longmapsto St', \text{then } \mathcal{P} \rhd St, St'' \longmapsto St', St'' \text{ for any } St''.$$

This property, typical of traditional multiset rewriting, allows processing head atoms incrementally, both in a centralized [3] and in a distributed [9] setting. Incremental processing is precisely what is done by the node-centric rules $[\![R]\!]$ a system-centric rule R is compiled into: a primary location combines data incrementally from the forwarding locations.

However, because comprehension patterns have a maximal semantics, monotonicity does not hold for full Comingle [6]. A naive approach to incrementally matching the head of a system-centric rule, as adapted from [9] for example, would be unsound. Consider the rule head $[X]p(Y_1, Y_2), \wr[Y_1]q(\vec{x})\int_{\vec{x}\to ts_1}, \wr[Y_2]q(\vec{x})\int_{\vec{x}\to ts_2}$ where incremental execution proceeds from left to right, say. By the time X has received the facts collected at Y_2, new facts matching $[Y_1]q(\vec{x})$ may have arrived at Y_1, violating maximality. We recover soundness by locking all facts that can thus compromise incremental processing. In general, these are facts headed by a predicate p such that the atom $[\ell]p(\vec{x})$ occurs as the subject F of a comprehension $\wr F \mid g\int_{\vec{x}\to ts}$ anywhere in the program. We call them *non-monotonic predicates*. Predicates that never appear within a comprehension pattern are *monotonic*, and we do not need to take special precautions for them.

5.1 An Example

As an example, consider the following Comingle rule, which we call swp:

$$\forall \left[\begin{array}{l} [X]swap(Y,P),\ \ [Y]okSwap \\ \wr[X]data(N) \mid N \leq P \int_{N \to Ns} \\ \wr[Y]data(M) \mid M \geq P \int_{M \to Ms} \end{array}\right] \multimap \left[\begin{array}{l} \wr[X]data(M) \int_{M \leftarrow Ms} \\ \wr[Y]data(N) \int_{N \leftarrow Ns} \end{array}\right]$$

This rule lets two parties X and Y atomically swap values up to a threshold P. It is triggered when node X holds a fact $swap(Y,P)$ while node Y holds $okSwap$. It retrieves all the facts $data(N)$ held at X such that $N \leq P$ (that is $\wr[X]data(N) \mid N \leq P \int_{N \to Ns}$) and sends them to Y (with body expression $\wr[Y]data(N) \int_{N \leftarrow Ns}$). At the same time, it transfers all $data(M)$ such that $M \geq P$ from Y (i.e., $\wr[Y]data(M) \mid M \geq P \int_{M \to Ms}$) to X (as $\wr[X]data(M) \int_{M \leftarrow Ms}$). The mention of Y as an argument of $swap$ makes this rule 1-neighbor restricted with X as its primary location and Y the only forwarding location.

This rule is compiled into the six node-centric rules (exec_X^{swp} to abort_X^{swp}) discussed next. Each of these rules executes an aspect of the overall system-centric rewriting embodied by rule swp. It makes use of various auxiliary predicates, which we capitalize for ease of identification, and it introduces new variables, which we write in lower case. We write the auxiliary predicates as a root possibly superscripted by a rule or predicate name, and possibly subscripted by a relevant location variable, for example Req_Y^{swp} below. We further highlight them using various background colors, that the reader may safely ignore. The new facts are categorized as follows.

- *Locking facts* have the form $[X]Free^p$. For emphasis, we will highlight locking facts with a light-blue background. Such facts are a means to lock non-monotonic predicates p in order to guarantee maximality: a rule that makes use of such a predicate at some location X, either in its head or in its body, will be compiled into a rule that acquires $[X]Free^p$, thereby inhibiting the execution of other rules that make use of p at X. This fact is put back into X's local state once the rule execution has completed successfully, or if it gets aborted.
- *Transaction facts* are of the form $[X]Next(n)$, $[X]Trans(e)$, $[X]Done(e)$ or $[X]Abort(e)$. We highlight them in pale orange. Their purpose is to keep track of and manage ongoing system-centric rule execution attempts, which we call *transactions*. The variable n is a counter incremented each time node X initiates a transaction, while e is another number computed from n and the location name X to act as a global transaction identifier. The fact $[X]Next(n)$ holds the current value of X's counter n, the fact $[X]Trans(e)$ indicate that transaction e is ongoing at X, while $[X]Done(e)$ and $[X]Abort(e)$ signal that e has either completed successfully or is being aborted.
- There are three types of *staging facts* for each rule R (identified by some unique name r), all highlighted in a pale green background for ease of identification. With the fact $[Y]Req^r_Y(e,X,\vec{x})$, primary location X issues a request to Y to gather relevant local facts in the head of R as part of transaction e. The parameters $\vec{x}$ list the information that X was able to secure and that may be useful to Y. The answer $\vec{y}$ is returned to X by means of the fact $[X]Ans^r_Y(e,Y,\vec{y})$. Finally, X can

remember information $\vec{z}$ for its own records by means of the fact $[X]\,Wait^{r}(\vec{z})$. They are used to implement the various stages of a two-phase commit among the parties involved.

The first compiled node-centric rule is to be executed at the primary location, X:

$$\forall \left[\begin{array}{l} [X]swap(Y,P), \\ \wr[X]data(N) \mid N \leq P \int_{N \to Ns}, \\ [X]Free^{data}, [X]Next(n) \end{array}\right] \multimap \left[\begin{array}{l} [Y]Req_{Y}^{swp}(e,X,Ns,P), \\ [X]\,Wait^{swp}(e,Y,Ns,P), \\ [X]\,Trans(e), [X]Next(n') \end{array}\right] \ (\mathsf{exec}_{X}^{swp})$$

where $e = H(X,n)$ and $n' = n+1$.

The head of this rule contains all the expressions that our original rule could match locally, namely $[X]swap(Y,P)$ and $\wr[X]data(N) \mid N \leq P \int_{N \to Ns}$. Because predicate $data$ occurs within a comprehension — it is non-monotonic — this rule also acquires a lock on it ($[X]Free^{data}$). Finally, it increments the local counter n (retrieved as $[X]Next(n)$ and reasserted as $[X]Next(n')$ with $n' = n+1$). The function $H(X,n)$ combines the value of this counter and the primary location's identity into a globally unique value e which will act as a transaction identifier, recorded as fact $[X]\,Trans(e)$. The body of this rule also includes the staging fact $[Y]Req_{Y}^{swp}(e,X,Ns,P)$ to request the matching data values from node Y. Note that the arguments mention the transaction identifier e, who to return the results to (X), and the variables corresponding to data that X could compute locally (a more refined compilation scheme could optimize Ns away as it is not needed by Y). Node X also asserts the staging fact $[X]\,Wait^{swp}(e,Y,Ns,P)$ for its own records, so that it can continue execution once it receives a response from Y.

The forwarding location Y can respond to X in one of two ways: by returning the requested data, or by aborting the transaction. A successful response begins with the following rule:

$$\forall \left[\begin{array}{l} [Y]okSwap, \ \wr[Y]data(M) \mid M \geq P \int_{M \to Ms}, \\ [Y]Free^{data}, [Y]Req_{Y}^{swp}(e,X,Ns,P) \end{array}\right] \multimap \left[\begin{array}{l} [Y]\,Trans(e), \\ [X]Ans_{Y}^{swp}(e,Y,Ms) \end{array}\right] \ (\mathsf{exec}_{Y}^{swp})$$

Here, Y retrieves its part of the original rule head, $\wr[Y]data(M) \mid M \geq P \int_{M \to Ms}$ and $[Y]okSwap$, and locks the non-monotonic predicate $data$ (with $[Y]Free^{data}$). It notes that it is engaged in transaction e with the fact $[Y]\,Trans(e)$ and sends X the expected answer, $[X]Ans_{Y}^{swp}(e,Y,Ms)$. Observe that, at this point, it is still in the transaction.

Next, X resumes execution by asserting the body of swp:

$$\forall \left[\begin{array}{l} [X]\,Wait^{swp}(e,Y,Ns,P), \\ [X]Ans_{Y}^{swp}(e,Y,Ms) \end{array}\right] \multimap \left[\begin{array}{l} \wr[X]data(M) \int_{M \leftarrow Ms}, \ \wr[Y]data(N) \int_{N \leftarrow Ns}, \\ \wr[l]Free^{data} \int_{l \leftarrow \wr X,Y \int}, \wr[l]Done(e) \int_{l \leftarrow \wr X,Y \int} \end{array}\right] \ (\mathsf{succ}_{X}^{swp})$$

With it, X combines the values it had computed locally ($[X]\,Wait^{swp}(e,Y,Ns,P)$) and the values obtained from Y (as $[X]Ans_{Y}^{swp}(e,Y,Ms)$) and asserts the body of the original rule ($\wr[X]data(M) \int_{M \leftarrow Ms}, \wr[Y]data(N) \int_{N \leftarrow Ns}$). It also releases all locks ($\wr[l]Free^{data} \int_{l \leftarrow \wr X,Y \int}$) and signals that the transaction has completed successfully ($\wr[l]Done(e) \int_{l \leftarrow \wr X,Y \int}$).

One last clean-up rule is needed to remove all facts associated with a completed transaction e, namely $\wr[Z]\mathit{Trans}(e)\int$, $[Z]\mathit{Done}(e)$. It does this at every participating node Z.

$$\forall\left(\wr[Z]\mathit{Trans}(e)\int, [Z]\mathit{Done}(e) \multimap \varnothing\right) \ (\mathsf{done})$$

The transaction started by rule (exec_X^{swp}) can fail for one of two reasons: either because the forwarding node Y does not have the requested data (e.g., if there is no okSwap at Y), or because Y is already engaged in possibly conflicting transactions. Although comprehension patterns are able to express the absence of a fact (or class of facts) in the state, we will approximate the first option by non-deterministically aborting the transaction (see the note below). Transaction failure is then captured by the following rule, executed at Y:

$$\forall\left(\wr[Y]\mathit{Trans}(e')\int_{e'\to es} \setminus [Y]\mathit{Req}_Y^{swp}(e,X,Ns,P) \mid e \mathrel{\dot{\ll}} es \multimap [X]\mathit{Abort}(e)\right) (\mathsf{fail}_Y^{swp})$$

where $e \mathrel{\dot{\ll}} es$ iff $es = \varnothing$ or for some $e' \in es$ and $e < e'$

Upon receiving the staging fact $[Y]\mathit{Req}_Y^{swp}(e,X,Ns,P)$, node Y collects all of its active transactions in the multiset es. The guard $e \mathrel{\dot{\ll}} es$ succeeds in one of two circumstances. The first is when there is no other ongoing transaction (which approximates an unsuccessful match). The second is when some other ongoing transaction e' has a larger identifier ($e < e'$). This guarantees that at least one transaction (the "strongest") will delay its decision to abort, until all others at the same location have terminated (with either success or failure). This avoids livelocks between transactions attempting to acquire the same facts. Conversely, the uniqueness of transaction identifiers guarantees that only one such transaction at a location delays its abort — otherwise we risk inducing deadlocks. Because rules (exec_X^{swp}) and (fail_Y^{swp}) are competing for the same staging fact $[Y]\mathit{Req}_Y^{swp}(e,X,Ns,P)$, exactly one of them is applicable in general, and only the latter is enabled when Y does not have the data requested by X.[1]

Rule (fail_Y^{swp}) is followed by rule (abort_Y^{swp}), examined next. It is executed at X:

$$\forall\left[\begin{array}{l}[X]\mathit{Trans}(e), [X]\mathit{Abort}(e),\\ [X]\mathit{Wait}^{swp}(e,Y,Ns,P)\end{array}\right] \multimap \left[\begin{array}{l}[X]\mathit{swap}(Y,P), \wr[X]\mathit{data}(N)\int_{N\leftarrow Ns},\\ [X]\mathit{Free}^{data}\end{array}\right] (\mathsf{abort}_Y^{swp})$$

It aborts transaction e by consuming the fact $[X]\mathit{Trans}(e)$ and reverting X's local computation, recorded in $[X]\mathit{Wait}^{swp}(e,Y,Ns,P)$), back into the state.

5.2 Choreographic Compilation

We now describe Comingle's choreographic compilation scheme on the basis of this intuition for one form of rules — see [7] for the general case. We write $NM(\mathcal{P})$ for the

[1] A version of rule (fail_Y^{swp}) that is mutually exclusive with rule (exec_X^{swp}) is as follows:

$$\forall\left(\wr[Y]\mathit{Trans}(e')\int_{e'\to es} \setminus [Y]\mathit{Req}_Y^{swp}(e,X,Ns,P), \underline{\wr[Y]\mathit{okSwap}\int_{()\to os}} \mid e \mathrel{\dot{\ll}} es \wedge \underline{os = \varnothing} \multimap [X]\mathit{Abort}(e)\right)$$

where the underlined components check that Y does not hold a fact okSwap. A general treatment of *negation as absence*, as this feature is known, is beyond the scope of this paper.

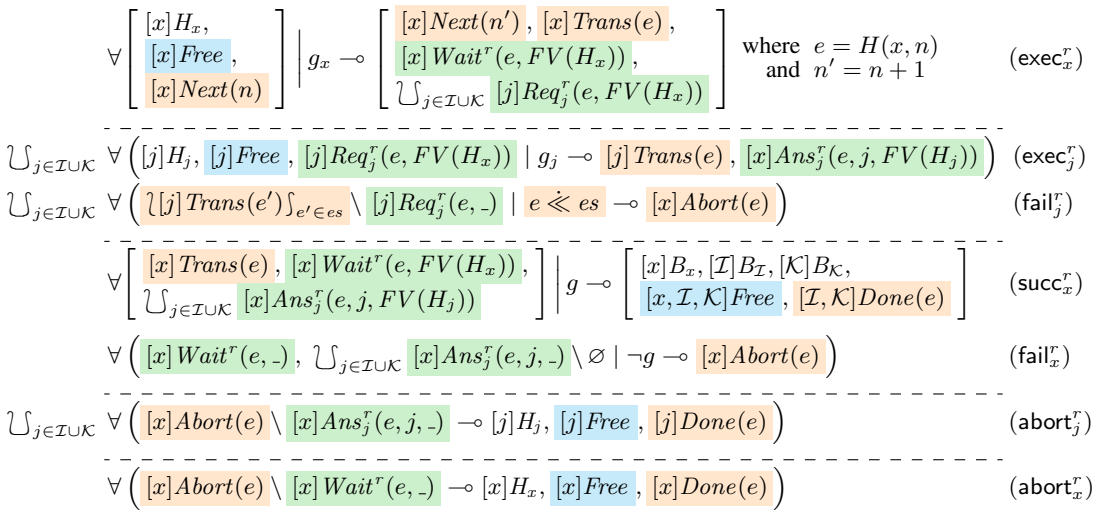

Fig. 3. Compilation of Simple Rule $\forall\, ([x]H_x,\ [\mathcal{I}]H_{\mathcal{I}}\ \mid\ g_x \wedge g_{\mathcal{I}} \wedge g\ \multimap\ [x]B_x,\ [\mathcal{I}]B_{\mathcal{I}},\ [\mathcal{K}]B_{\mathcal{K}})$

set of all non-monotonic predicate names in program $\mathcal{P}$ and $NM_{\mathcal{P}}(\overline{E})$ for the subset of $NM(\mathcal{P})$ that occur in expressions $\overline{E}$ (see [7] for a formal definition).

The choreographic compilation $\llbracket \mathcal{P} \rrbracket$ of a 1-neighbor restricted program $\mathcal{P}$ is a node-centric program equivalent to $\mathcal{P}$, a program that consists only of node-centric rules. The compiled program $\llbracket \mathcal{P} \rrbracket$ is comprised of the compilation $\llbracket R \rrbracket^{\mathcal{P}}$ of each source rule R in $\mathcal{P}$, plus rule (done) above — this rule is "global" in that it is shared by the encoding of all rules in $\mathcal{P}$.

$$\llbracket \mathcal{P} \rrbracket \;=\; \begin{cases} \bigcup_{R\in\mathcal{P}} \llbracket R \rrbracket^{\mathcal{P}} & (\textit{rule names given below}) \\ \forall j, e.\, \wr\, [j]Trans(e) \int, [j]Done(e) \multimap \varnothing & (\mathsf{done}) \end{cases}$$

Simple 1-Neighbor Restricted Rules. We consider *simple* 1-neighbor restricted rules, that contain only *localized comprehension patterns*. The location of the subject of such patterns is bound outside the comprehension itself. Thus, in rule (2), the comprehension $\wr [X] neighbor(N) \int_{N\to Ns}$ was localized, but $\wr [N] temp(T) \mid N \in Ns \int_{T\to Ts}$ is not. The case study in Section 5.1 consisted of a simple rule.

Using some abbreviations of convenience (explained next), a source rule R with such characteristics can be written as follows:

$$\forall x, \mathcal{I}, \mathcal{K}.\, [x]H_x,\ [\mathcal{I}]H_{\mathcal{I}}\ \mid\ g_x \wedge g_{\mathcal{I}} \wedge g\ \multimap\ [x]B_x,\ [\mathcal{I}]B_{\mathcal{I}},\ [\mathcal{K}]B_{\mathcal{K}}$$

Here, x is the primary location of R and H_x collates all the facts in R's head located at x. These can be either atoms $[x]f$ or localized comprehensions $\wr [x]f \mid g \int_{z\to zs}$. Similarly, B_x refers to the body expressions located at x. The set $\mathcal{I}$ contains all forwarding nodes that locate facts in the head of R. We write $[\mathcal{I}]H_{\mathcal{I}}$ for $\bigcup_{i\in\mathcal{I}} H_i$ where each H_i

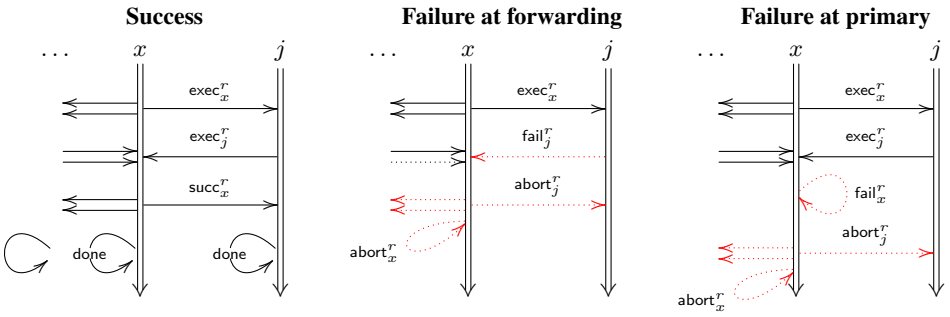

Fig. 4. Possible Execution Sequences of a Compiled System-centric Rule

follows the same conventions as H_x, and similarly for body expressions $B_{\mathcal{I}}$. The set $\mathcal{K}$, disjoint from x and $\mathcal{I}$, lists all forwarding nodes that locate expressions only in the body of R — we call them *receiving locations*. The guard of R is partitioned into fragment g_x whose satisfiability can be determined locally by x, i.e., such that $FV(H_x) \stackrel{g_x}{\Longrightarrow} \vec{y}$ for some $\vec{y}$, into similar fragments g_i for each $i \in \mathcal{I}$ which we abbreviate as $g_{\mathcal{I}}$, and into a final fragment g which cannot be determined locally by any of these nodes.

Figure 3 defines the compilation $[\![\mathcal{P}]\!]^R$ of a simple system-centric rule R named r. The constituent node-centric rules implement a two-phase commit with the same behavior as R. The possible executions patterns are sketched in Figure 4. The encoding $[\![\mathcal{P}]\!]^R$ relies on further abbreviations: given a location j among $x, \mathcal{I}, \mathcal{K}$, we write $[j]Free$ for $\bigcup_{p \in NM_{\mathcal{P}}(H_j, B_j)} [j]Free^p$, the locks of all non-monotonic predicates mentioned by j. Furthermore, we write $[\mathcal{J}]Free$ for $\bigcup_{j \in \mathcal{J}} [j]Free$, the locks of any location j in set $\mathcal{J}$. We also write "_" for irrelevant terms, realized in a rule as an appropriate number of single-occurrence variables.

Rule (exec$_x^r$) initiates execution at the primary location x by matching local facts ($[x]H_x$), locking non-monotonic predicates ($[x]Free$) and setting up a new transaction e as discussed in Section 5.1. It sends a request to each forwarding location ($\bigcup_{j \in \mathcal{I} \cup \mathcal{K}} [j]Req_j^r(e, FV(H_x))$) and prepares for their response with $[x]Wait^r(e, FV(H_x))$. Rule (exec$_j^r$) implements a successful reply by a forwarding location j: it locks its non-monotonic predicates ($[j]Free$), records the transaction ($[j]Trans(e)$) and returns a response ($[x]Ans_j^r(e, j, FV(H_j))$) — in the case of a receiving location in $\mathcal{K}$ this amounts to just locking non-monotonic predicates. Execution then continues at x with rule (succ$_x^r$) which collates all the responses ($\bigcup_{j \in \mathcal{I} \cup \mathcal{K}} [x]Ans_j^r(e, j, FV(H_j))$), checks the remaining guards g, rolls out the body of R, frees all non-monotonic predicates and prepares for clean-up using rule (done). This execution sequence is shown on the left of Figure 4.

As described in Section 5.1, rule (fail$_j^r$) aborts execution at a forwarding node j in response to a match failure or to preempt livelock. This is followed by rules (abort$_j^r$) and (abort$_x^r$) with which primary location x rolls back all head facts that had been

consumed and frees the locks. This possibility is sketched in the central portion of Figure 4.

One more behavior, which was not possible in the example in Section 5.1, is depicted on the right-hand side of Figure 4. Here, primary x has collected responses from all forwarding locations, but they cannot be combined as prescribed by R because the guard g is not satisfiable. In this case, rule (fail^r_x) is applicable and triggers the abort rules just described.

Generic 1-neighbor Restricted Rules. The techniques deployed for simple rules form the core of the compilation of generic 1-neighbor restricted rules whose full treatment can be found in [7], for space reasons. The treatment of system-centric comprehension patterns, whose subject $[z]f$ is located at a bound variable z, requires some care as such z may be among $\mathcal{I}$ or $\mathcal{K}$, or could be x itself above. Naively sending separate requests for different predicates would stall execution as the first request would lock the non-monotonic predicates, thereby preventing the others from making progress. We address this issue by having each node involved acquire all information it needs in one go.

Store. The node-centric encoding of a store St, denoted $\llbracket St \rrbracket^{\mathcal{P}}_{\vec{c}}$, extends St with, for each location ℓ in an ensemble $\mathcal{L}$, (a) a locking fact $[\ell]\mathit{Free}^p$ for each non-monotonic predicate p of $\mathcal{P}$ and (b) a local transaction counter c_ℓ in $\vec{c}$ for ℓ. It is defined as follows:

$$\llbracket St \rrbracket^{\mathcal{P}}_{\vec{c}} \;=\; \begin{cases} St, \\ \bigcup_{\ell \in \mathcal{L}} \bigcup_{p \in NM(\mathcal{P})} [\ell]\mathit{Free}^p, \\ \bigcup_{\ell \in \mathcal{L}} [\ell]\mathit{Next}(c_\ell) \end{cases}$$

6 Formal Results

In this section, we present some properties of the transformation in the previous section, and give sketches of their proof. Details can be found in [7]. Specifically, we show that choreographic compilation is sound and complete: it transforms a system-centric program into node-centric rules with what amounts to the same behavior. We also note that the computation carried out by a compiled rule can never get stuck midway.

For the convenience of the reader, we distinguish between source and encoded rewrite states by denoting the former St and the latter Et. The operation $\lVert Et \rVert^{\mathcal{P}}$ that *decodes* a compiled state Et back into a corresponding source state St, relative to a 1-neighbor restricted program $\mathcal{P}$ is defined in [7]. It does so by discarding all locking and transaction facts, by keeping source facts, and by reverting request and waiting facts to the source fact they had replaced. Other staging facts are ignored. We write $\mathit{obligations}(\mathcal{P}, E)$ for the result of this extraction on a staging fact E.

To begin with, the following property shows that for every reachable encoded state Et, we can always derive another state Et' such that it does not contain "stuck" transactions. Specifically, there are no encoded matching obligations in Et'.

Theorem 1 (Progress). *If* $\llbracket \mathcal{P} \rrbracket \rhd \llbracket St \rrbracket^{\mathcal{P}}_{\vec{c}} \longmapsto^* Et$, *then* $\llbracket \mathcal{P} \rrbracket \rhd Et \longmapsto^* Et'$ *for some* Et' *such that* $\mathit{obligations}(\mathcal{P}, Et') = \varnothing$.

Proof. By structural induction on our choreographic transformation schemes, we first show that any individual transaction e can always be concluded as an abort or a successful application of a system-centric rewriting. In either case, encoded matching obligations are consumed. This result is extended to an arbitrary number of transactions, which the antecedent of the theorem may be executing concurrently.

Next, we show that every derivation of a compiled program $[\![\mathcal{P}]\!]$ is derivable in its source states. Hence the choreographic transformation is sound.

Theorem 2 (Soundness). *If* $[\![\mathcal{P}]\!] \rhd [\![St]\!]^{\mathcal{P}}_{\vec{c}} \longmapsto^* Et$, *then* $\mathcal{P} \rhd St \longmapsto^* \lfloor\!\lfloor Et \rfloor\!\rfloor^{\mathcal{P}}$

Proof. The proof starts by considering a single step, where it induces the definition of our choreographic transformation. Specifically, we show that every encoded derivation step corresponds to either a source derivation step or a stuttering step (i.e., zero step $\mathcal{P} \rhd St \longmapsto^* St$). This involves showing that for derivable states of each choreographic transformation, every removal of a matching obligation (i.e., $[j]H_j$) is accompanied by an addition of a corresponding encoded matching obligation (e.g., $[j]\,Wait^r(e, j, \vec{v})$), hence via the decoding operation, source facts are never wrongfully omitted. A simple induction lifts this result to the multiple step case.

Theorem 3 states that every derivation of a source program $\mathcal{P}$ can be simulated by $[\![\mathcal{P}]\!]$. Hence the choreographic transformation is complete.

Theorem 3 (Completeness). *If* $\mathcal{P} \rhd St \longmapsto^* St'$, *then* $[\![\mathcal{P}]\!] \rhd [\![St]\!]^{\mathcal{P}}_{\vec{c}} \longmapsto^* Et'$ *for some* Et' *such that* $\lfloor\!\lfloor Et' \rfloor\!\rfloor^{\mathcal{P}} = St'$.

Proof. The proof proceeds by structural induction on our transformation. Specifically, we show that, using Theorem 1, we can always simulate each source derivation step with a series of encoded derivation steps that applies a transaction of the corresponding source derivation. Therefore, Et' exists. A further induction is used to stretch this result to multiple derivation steps.

7 Related Works

An extension of Datalog for implementing network protocols is explored in [12]. This paper defines *link-restricted* Datalog rules and *rule localizing* encodings which are specific instances of neighbor restriction and choreographic transformation discussed here.

Our work draws inspiration from research on choreographic programming (e.g., [10]). An example of a coordination language in this domain is Jolie [11], which is targeted at service-oriented web applications. By and large, these works focus on choreographic projections of lower-level imperative-style programming languages, our transformation share the same goals and intuition, at a higher level of abstraction, though.

The execution model underlying Comingle is inspired by the run-time architecture of Constraint Handling Rules [3] (CHR). Also based on rule-based multiset rewriting, the CHR language can be viewed as an ancestor of Comingle. There is abundant research on exploiting CHR as a parallel execution model, of example as an extension of the actor model [13] and for programming FGPAs [14].

Comingle is a logic programming framework aimed at simplifying the development of applications distributed over multiple mobile devices. The original prototype [5,8] targeted the Android SDK, and has recently been extended to x86 devices running Java, thereby supporting mobile applications over heterogeneous platforms.

8 Conclusions

In this paper, we develop a choreographic compilation scheme for multiset rewriting languages with support for multiset comprehension patterns. This choreographic compilation scheme preserves soundness: it transforms a system-centric Comingle program into a node-centric encoding that ensures both atomicity of the rule application and maximality of comprehension patterns. Node-centric encodings have a straightforward node-centric operational interpretations (message passing), and thus are immediately executable by individual computing nodes. In all, our work here provides a foundational bridge between high-level system-centric specifications of decentralized multiset rewriting with comprehension patterns, and their lower-level node-centric operational interpretations.

References

1. M. P. Ashley-Rollman et al. A Language for Large Ensembles of Independently Executing Nodes. In *ICLP'09*, pages 265–280. Springer LNCS 5649, 2009.
2. L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR 2010*, pages 222–236, Paris, France, 2010. Springer LNCS 6269.
3. T. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.
4. S. Grumbach and F. Wang. Netlog, a Rule-based Language for Distributed Programming. In *PADL'10*, pages 88–103. Springer LNCS 5937, 2010.
5. E. S. Lam. CoMingle: Distributed Logic Programming Language for Android Mobile Ensembles. Download at `https://github.com/sllam/comingle`, 2014.
6. E. S. Lam and I. Cervesato. Optimized Compilation of Multiset Rewriting with Comprehensions. In *APLAS'14*, pages 19–38, Singapore, 2014. Springer LNCS 8858.
7. E. S. Lam and I. Cervesato. Decentralized Compilation of Multiset Comprehensions. Technical Report CMU-CS-16-101, Carnegie Mellon University, 2016.
8. E. S. Lam, I. Cervesato, and N. F. Haque. Comingle: Distributed Logic Programming for Decentralized Mobile Ensembles. In *COORDINATION'15*. Springer LNCS 9037, 2015.
9. E. S. L. Lam and I. Cervesato. Decentralized execution of constraint handling rules for ensembles. In *PPDP'13*, pages 205–216, Madrid, Spain, 2013.
10. I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *SEFM '08*, pages 323–332, 2008.
11. I. Lanese, F. Montesi, and G. Zavattaro. The Evolution of Jolie — From Orchestrations to Adaptable Choreographies. In *Software, Services, and Systems*, pages 506–521, 2015.
12. B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *SIGMOD'06*, pages 97–108, 2006.
13. M. Sulzmann, E. S. L. Lam, and P. V. Weert. Actors with multi-headed message receive patterns. In *COORDINATION 2008*, pages 315–330, 2008.
14. A. Triossi, S. Orlando, A. Raffaetà, and T. Frühwirth. Compiling CHR to Parallel Hardware. In *PPDP'12*, pages 173–184, New York, NY, USA, 2012.