Decentralized Execution of Constraint Handling Rules for Ensembles *

Edmund S. L. Lam
Carnegie Mellon University
sllam@qatar.cmu.edu

Iliano Cervesato
Carnegie Mellon University
iliano@cmu.edu

Abstract

CHR is a declarative, concurrent and committed choice rule-based constraint programming language. In this paper, we adapt CHR to provide a decentralized execution model for parallel and distributed programs. Specifically, we consider an execution model consisting of an ensemble of computing entities, each with its own constraint store and each capable of communicating with its neighbors. We extend CHR into CHR^e, in which rules are executed at a location and are allowed to access the constraint store of its immediate neighbors. We give an operational semantics for CHR^e, denoted ω_0^e, that defines incremental and asynchronous decentralized rewriting for the class of CHR^e rules characterized by purely local matching (0-neighbor restricted rules). We show the soundness of the ω_0^e semantics with respect to the abstract CHR semantics. We then give a safe encoding of the more general 1-neighbor restricted rules as 0-neighbor restricted rules, and discuss how this encoding can be generalized to all CHR^e programs.

Categories and Subject Descriptors F.3.2 [*Theory of Computation*]: Logics and Meanings of Programs—Semantics of Programming Languages

General Terms Languages, Performance, Theory

Keywords Distributed Programming, Constraint Logic Programming, Multiset Rewriting

1. Introduction

In recent years, we have seen many advances in distributed systems, multicore architectures and cloud computing, drawing more research interest into better ways to harness and coordinate the combined power of distributed computation. While this has made distributed computing resources more readily accessible to mainstream audiences, the fact remains that implementing distributed applications that can exploit such resources via traditional distributed programming methodologies is an extremely difficult task. As such, finding effective means of programming distributed systems is more than ever an active and fruitful research and development endeavor.

* Funded by the Qatar National Research Fund as project NPRP 09-667-1-100 (Effective Programming for Large Distributed Ensembles)

PPDP'13, September 16–18, 2013, Madrid, Spain.

$$\begin{aligned}
&\mathit{base} : [X]\,\mathit{edge}(Y, D) \Longrightarrow [X]\,\mathit{path}(Y, D)\\
&\mathit{elim} : [X]\,\mathit{path}(Y, D) \setminus [X]\,\mathit{path}(Y, D') \Longleftrightarrow D < D' \mid \mathit{true}\\
&\mathit{trans} : [X]\,\mathit{edge}(Y, D), [Y]\,\mathit{path}(Z, D')\\
&\qquad\qquad \Longrightarrow X \neq Z \mid [X]\,\mathit{path}(Z, D + D')
\end{aligned}$$

Figure 1. Distributed All Shortest Path

In this paper, we propose an extension of the constraint programming language CHR [5] (Constraint Handling Rules). The resulting language, which we call CHR^e is designed specifically as a high-level distributed and parallel programming language for developing applications that operate in a decentralized manner over an *ensemble* of distributed computing entities, referred to as locations. Each CHR rule executes at a location, enabling this location to read and write data held by its immediate neighbors. Specifically, we are interested in rules that can read data from up to n of their immediate neighbors for various values of n, but writes to any number of neighbors. We call them n*-neighbor restricted rules*. Such n-neighbor restricted rules are a generalization of link restriction [1, 8] adapted to the context of multiset rewriting. This gives us a highly expressive model for programming complex behaviors involving distributed ensembles in a declarative and concurrent manner.

Figure 1 shows an example CHR^e program that contains three CHR rules. This program computes all shortest paths of a graph in a distributed manner. An edge of length D from a location X to Y is represented by a constraint $\mathit{edge}(Y, D)$ found in X's constraint store, written $[X]\,\mathit{edge}(Y, D)$. Similarly $[X]\,\mathit{path}(Y, D)$ expresses a path of length D from X to Y. The location of a constraint is modeled by the $[l]$ operator that prefixes all constraints in the CHR rules. The rules base and elim are 0*-neighbor restricted* rules because their left-hand sides involve constraints from exactly one location, X. Rule trans is a 1-neighbor restricted rule since its left-hand side involves X and a neighbor Y. We designate X as the *primary* location of this CHR rule because it references Y in an argument (here, $[X]\,\mathit{edge}(Y, D)$). This means that Y is a topological neighbor of X[1]. Neighbor restriction brings aspects of the topology of the ensemble (X has an edge to Y) in the CHR rules. Rule base adds $\mathit{path}(Y, D)$ at location X for every instance of constraint $\mathit{edge}(Y, D)$ at the same location. Rule elim looks for any instances of a pair of constraints, $\mathit{path}(Y, D)$ and $\mathit{path}(Y, D')$ involving the same location Y at location X and removes the longer of the two path ($\mathit{path}(Y, D')$ for $D < D'$). Rule trans adds $\mathit{path}(Z, D + D')$ at location X whenever there is $\mathit{edge}(Y, D)$ at location X and a matching $\mathit{path}(Z, D')$ at location Y for $X \neq Z$. This program is *declarative* because the pro-

[1] Any location reference Y that appears as an argument of a constraint at location X is considered a neighbor of X. This is how neighbor restriction enforces topological information to be embedded in predicates.

grammer focuses on *which* distributed computations to synchronize (e.g., $[X]\, edge(Y, D)$, $[Y]\, path(Z, D')$) to produce *what* results ($[X]\, path(Z, D + D')$), rather than *how* synchronization is achieved. It is *concurrent* because while a CHR rule applies to a fragment of the ensemble, many other rewritings can occur asynchronously in the rest of the ensemble.

In this paper, we present CHR^e, a distributed programming language that extends CHR with *located constraints* and *n-neighbor restricted rules*, a generalized notion of link-restriction [8]. We define an abstract semantics of CHR^e, written ω_α^e, and prove its soundness with respect to the standard semantics of CHR. Following this, we extend the original CHR refined operational semantics [3] to support decentralized incremental multiset matching for 0-neighbor restricted rules, obtaining the ω_0^e operational semantics, and prove its soundness and the exhaustiveness of rule application, with respect to the ω_α^e semantics. We then give an optimized encoding of 1-neighbor restricted rules into 0-neighbor restricted rules of ω_0^e, prove the soundness of this encoding. Following this, we briefly discuss the generalization of this encoding to n-neighbor restricted rules.

The main contributions in this paper are as follows:

- We present CHR^e, a distributed programming language that extends CHR with *located constraints* and *n-neighbor restricted rules*, a generalized notion of link-restriction [8] in the context of multiset rewriting.
- We define the ω_α^e abstract semantics of CHR^e and prove its soundness with respect to the CHR abstract semantics.
- We present the ω_0^e operational semantics (based on the refined CHR operational semantics [3]) that supports decentralized incremental multiset matching for 0-neighbor restricted rules. We prove its soundness with respect to the ω_α^e abstract semantics.
- We give an optimized encoding of 1-neighbor restricted rules in ω_0^e and prove the soundness of this encoding.

The rest of this paper is organized as follows: We review related work in Section 2. Section 3 recalls notions used in this paper and the traditional CHR language. Section 4 defines the CHR^e language while Section 5 presents two examples of programming in CHR^e. Section 6 gives the ω_α^e abstract semantics, the ω_0^e operational semantics and relates them. Section 7 encodes 1-neighbor restricted rules into ω_0^e. In Section 8, we outline encoding of the full CHR^e language (n-neighbor restricted rules) into ω_0^e. We conclude in Section 9. Details of the generalized encoding, as well as proofs of all lemmas and theorems in this paper can be found in the extended technical report [6].

2. Related Works

An extension of Datalog for implementing network protocols is introduced in [8]. It defines *link restricted Datalog rules* along with the idea of *rule localization* which encodes link restricted rules into local Datalog rules. Our work adapts, expands and generalizes these ideas to the context of distributed multiset rewriting. A *distributed* and *incremental* algorithm for computing rule matchings involved in such distributed Datalog systems is presented in [10] and is analogous to what ω_0^e provides in the context of distributed multiset rewriting. Our work draws inspiration from the programming language Meld [1]. Meld introduces a multitude of extensions to Datalog, including linearity, aggregates and comprehensions. Work in [2] adapts Meld for the context of general purpose multicore parallel programming. The current implementation of Meld does not exploit the incremental nature of multiset matching[2], hence our work on ω_0^e can be adapted as an alternative operational semantics for Meld that does that. The ω_0^e semantics extends the refined CHR operational semantics [3]. Our work shares some similarities with works on parallel compilations of CHR [7, 15], but differs in that we focus on explicitly distributing the constraint store.

3. Preliminaries

In this section, we present the notations that will be used throughout this paper. We also give a brief introduction to the abstract CHR language and its semantics.

Let o be a generic syntactic construct or runtime object of our language. We write $\bar{o}$ for a multiset of objects o and $\vec{o}$ for a sequence of objects o. Relying on these notational accents for disambiguation, we write ',' to represent both multiset union and sequence concatenation, with $\emptyset$ as the identity element. For instance, '$o, \bar{o}$' is a multiset while '$o, \vec{o}$' is a sequence (the former is commutative while the latter is not). A set is treated as a multiset with single occurrences of each element. We use standard notations for sets: $\bar{o}_1 \cup \bar{o}_2$ for union, $o \in \bar{o}$ for membership, and $\bar{o}_1 \subseteq \bar{o}_2$ for subset. Given a set $\mathcal{I}$ of labels, we write $\biguplus_{i \in \mathcal{I}} o_i$ to denote the multiset of objects o_i, for $i \in \mathcal{I}$.

The substitution of all occurrences of variable x in o with the term expression t is denoted as $[t/x]o$. It is inductively defined on all syntactic constructs and runtime objects o in the usual manner. We assume that substitution is capture-avoiding and rely on implicit α-renaming. Given a sequence of terms $\vec{t}$ and a sequence of variables $\vec{x}$, we write $[\vec{t}/\vec{x}]o$ as the simultaneous substitution of each variable in $\vec{x}$ with the corresponding term in $\vec{t}$. We will use this notion with sets as well (i.e., $\bar{t}$ and $\bar{x}$). We write $\mathrm{FV}(o)$ for the set of free-variables in o and say that o is *ground* if $\mathrm{FV}(o) = \emptyset$. We write meta level operators in upright font (e.g., $\mathrm{FV}(-)$) and CHR constraint predicates in italics (e.g., $edge$, $path$).

Figure 2 illustrates the abstract syntax of CHR. CHR is a high-level multiset rewriting language built on top of a *term language*. As done traditionally [5], we will keep this term language mostly abstract but with some basic assumptions: We assume that it has variables and that each term t has a normal form, denoted $\mathrm{NF}(t)$. Well-formed term expressions can contain pure function applications[3] which we assume will evaluate to normal form terms (via $\mathrm{NF}(t)$). A rule guard G is a set of relations among term expressions, called *built-in constraints*. We assume that built-in constraints contain equality. The judgment $\models G$ decides the validity of ground built-in constraint G.

A CHR constraint $p(\vec{t})$ is a first-order predicate symbol p applied to a sequence of terms $\vec{t}$. A CHR rule is of the form $r : P \setminus S \iff G \mid B$ where r is a unique name, P, S and B are multisets of CHR constraints and G is a set of built-in constraints. Each CHR rule specifies a rewriting that can occur over fragments of a multiset of constraints, known as the *CHR store*. Specifically, a rule like this states that we can replace any matching instance of 'P, S' in the CHR store with the corresponding instance of B, if G is valid when applied to the matching substitution ($\models \theta G$). We will refer to P and S as the *propagate* and *simplify* rule heads, G as the rule *guard* and B as the *rule body*. This general form of CHR rule is known as a *simpagation rule*. The short-hands $r : P \implies G \mid B$ and $r : S \iff G \mid B$, known as *propagation* and *simplification* rules, have empty simplified and propagated rule heads respectively. We will omit the rule guard component if it is empty as well. We take a few benign deviations from traditional treatments of CHR (e.g., [5]): we require that all constraints in a CHR store be ground and that built-in constraints only appear in the guards G. This allows us to focus on the distributed multiset rewriting problem in this paper and set aside features like explicit built-in constraints as orthogonal extensions. Also, a rule body B can be prefixed by zero or more existential

[2] Rule matchings are re-computed after application of a rule instance.

[3] In Section 5, we will define these term functions when required.

Variables x Values v Predicate names p Rule names r Rule guard G

Term	t	::=	$x \mid v \mid \ldots$				
Constraint	c	::=	$p(\vec{t})$	CHR Rule	R	::=	$r : H \setminus H \iff G \mid B$
Rule Heads	H	::=	$\cdot \mid c, H$	Program	$\mathcal{P}$	::=	$R \mid R\,\mathcal{P}$
Rule Body	B	::=	$\exists \bar{x}.\, D$	Store	$\bar{\mathcal{S}}$	::=	$\emptyset \mid \bar{\mathcal{S}}, c$
Rule Body Elements	D	::=	$true \mid c, D$				

$$\frac{r : P' \setminus S' \iff G \mid B \in \mathcal{P} \qquad \models \theta G \qquad P = \theta P' \qquad S = \theta S'}{\mathcal{P} \;\rhd\; (\bar{\mathcal{S}}, P, S) \;\mapsto_{\omega_\alpha}\; (\bar{\mathcal{S}}, P, \mathrm{NF}(\mathrm{Inst}(\theta B)))}$$

Figure 2. Constraint Handling Rules, Language and Semantics

variable declarations ($\exists \bar{x}$). Such variables, which do not appear in the scope of the rule heads, are instantiated to new constants (that do not appear in the store) during rule application, allowing the creation of new *destinations* in destination passing-style programming [11]. For a rule body $\exists \bar{x}.\, D$ such that $\bar{x}$ is an empty set, we simply write it as D. A CHR rule $r : P \setminus S \iff G \mid B$ is *well-formed* if rule guards and rule body are grounded by the rule heads ($\mathrm{FV}(G) \cup \mathrm{FV}(B) \subseteq \mathrm{FV}(P) \cup \mathrm{FV}(S)$[4]) and all term expressions that appear in the rule are well-formed. A CHR program $\mathcal{P}$ is *well-formed* if all CHR rules in $\mathcal{P}$ are well-formed and have a unique rule name. A CHR store $\bar{\mathcal{S}}$ is a multiset of constraints. It is *well-formed* if $\mathrm{FV}(\bar{\mathcal{S}}) = \emptyset$ and all terms that appear in constraints are well-formed.

We inductively extend the normalization function to rule bodies. Given a rule body that contains no existentials D, $\mathrm{NF}(D)$ denotes the normalized rule body with all term expressions in D in normal form. Since this operation only applies to rule bodies that contain no existentials, we define a complementary meta operator that instantiates existential variables to fresh constants: Given a rule body $\exists \bar{x}.D$, $\mathrm{Inst}(\exists \bar{x}.D)$ denotes an instance of D such that each existential variable $\bar{x}$ that appear in D is replaced by a fresh constant a.

Figure 2 also shows the abstract semantics ω_α of CHR. It defines the transitions of well-formed CHR stores via the derivation step $\mathcal{P} \rhd \bar{\mathcal{S}} \mapsto_{\omega_\alpha} \bar{\mathcal{S}}'$ for a given well-formed CHR program $\mathcal{P}$. A derivation step models the application of a CHR rule: it checks that rule heads P' and S' match fragments P and S of the store under a matching substitution θ and that the corresponding instance of guard G is valid ($\models \theta G$), resulting to the rewrite of S with θB with existential variables instantiated and then all term expressions evaluated to normal form (i.e., $\mathrm{NF}(\mathrm{Inst}(\theta B))$). Derivation steps preserve the well-formedness of constraint stores. We denote the reflexive and transitive application of derivation steps as $\mathcal{P} \rhd \bar{\mathcal{S}} \mapsto^*_{\omega_\alpha} \bar{\mathcal{S}}'$. A CHR store $\bar{\mathcal{S}}$ is *terminal* for $\mathcal{P}$ if no derivation steps of ω_α applies in $\bar{\mathcal{S}}$.

While the semantics in Figure 2 models CHR rewritings abstractly, actual CHR implementations implement some variant of the refined operational semantics [3]. This semantics computes new matches incrementally from newly added constraints. Additionally, rule heads are implicitly ordered by an *occurrence index* (typically in textual order of appearance) and matches are attempted in that order, thus providing programmatic idioms that assume a textual ordering of rule application. More details of these will be provided in Section 6.2.

4. The CHR^e Language

In this section, we introduce the CHR^e language. CHR^e extends CHR in several ways and here we focus on the syntactic ramifications of these extensions. *Locations* l in CHR^e rewriting rules are name annotations to a CHR constraint c, written as $[l]\,c$ and read as 'c is located at l'. We call $[l]$ the *localization operator*. Locations can be variables and in this case are subjected to substitution and other free variable meta operations as if they were an additional argument in constraints. Operationally, $[l]\,c$ indicates that constraint c is held at location l (details in Section 6.2). Figure 3 shows the abstract syntax of CHR^e rules and programs. All constraints in a rule are now explicitly localized by the operator $[l]$. A localization operator in a rule head indicates the location where the constraint is to be matched, while a localization operator in a rule body indicates the location where that constraint is to be delivered. We call the former locations *matching locations* of the rule while the latter are *forwarding locations*.

CHR^e rules and programs are *well-formed* similarly to CHR rules and programs. Like all variables, location variables of rule bodies must appear as term arguments or localization operators of rule heads, or otherwise be existentially quantified. These "existential locations" represent new locations to be created during rule application and allows the specification of dynamically growing ensembles. We will only consider well-formed CHR^e rules from now on. Given a multiset of located constraints H, $\mathrm{Locs}(H)$ denotes the set of locations that appear in the localization operators of H (e.g., $\mathrm{Locs}([X]\,a(Y,Z), [Y]\,b(Z,2)) = \{X, Y\}$) and $\mathrm{Args}(H)$ denotes the set of all term arguments of constraints in H (e.g., $\mathrm{Args}([X]\,a(Y,Z), [Y]\,b(Z,2)) = \{Y, Z, 2\}$). We write $H_{|l}$ for the *location restriction* on H, which denotes the multiset of all constraints in H that is located at l (e.g., $([X]\,a(Y,Z), [Y]\,b(Z,2))_{|X} = a(Y,Z)$). We write $[l]\,H$ for the multiset of all constraints in H each prefixed with $[l]$.

Link restriction in distributed rule based languages [2, 8] constrains how locations are used. We generalize it to the notion of n-neighbor restriction. A CHR^e rule $r : P \setminus S \iff G \mid B$ is *n-neighbor restricted* (where $n = |\mathrm{Locs}(P,S)| - 1$) if we can select $l \in \mathrm{Locs}(P,S)$ such that it is *directly connected* to each other n locations that appear in the rule heads. Furthermore, rule heads of locations other than l are *isolated* in that they do not contain common variables that do not appear at l. Rule guards also need to be *isolated* in a manner such that each atomic rule guard is grounded by the rule heads of location l and at most one other location of the rule head. Such a matching location l is called the *primary location* of the n-neighbor restricted rule, while all other matching locations of the rule are called *neighbor locations*. We assume that rule guards G have no side-effects and hence are pure boolean assertions. Formally, a CHR^e rule $r : P \setminus S \iff G \mid B$ is *n-neighbor restricted* (for $n = |\mathrm{Locs}(P,S)| - 1$) if there exists $l \in \mathrm{Locs}(P,S)$ such that the following three conditions are satisfied:

- *Directly connected:* $\mathrm{Locs}(P,S) \subseteq \{l\} \cup \mathrm{Args}((P,S)_{|l})$
- *Neighbor isolated rule heads:* For all other distinct $l', l'' \in \mathrm{Locs}(P,S)$, for each $x \in \mathrm{FV}((P,S)_{|l'}, (P,S)_{|l''})$, we have $x \in \mathrm{FV}((P,S)_{|l})$.

[4] Recall that the existential variables are not free.

Location names k

Location	l	::=	$x \mid k$	Rule Heads	H	::=	$\cdot \mid [l]\,c, H$
CHR Rule	R	::=	$r : H \setminus H \iff G \mid B$	Rule Body	B	::=	$\exists \bar{x}.D$
Program	$\mathcal{P}$	::=	$R \mid R\,\mathcal{P}$	Rule Body Elements	D	::=	$true \mid [l]\,c, D$

Figure 3. Abstract syntax of CHR^e

- *Neighbor isolated rule guards:* For each guard condition $g \in G$, either we have $\mathrm{FV}(g) \subseteq \mathrm{FV}((P,S)_{|l})$ or for some other $l' \in \mathrm{Locs}(P,S)$, we have $\mathrm{FV}(g) \in (\mathrm{FV}((P,S)_{|l}) \cup \mathrm{FV}((P,S)_{|l'}))$.

n-neighbor restricted rules characterizes multiset rewritings across constraint stores of $n+1$ connected locations in a 'star' topology with the *primary* location in the center, directly connected to each n neighbors. The neighbor isolation conditions (for rule heads and guards) are defined so as to make the matching problem decomposable into partial match problems between the primary location l and each of its n-neighbors separately (details in Section 8). In general, an n-neighbor restricted rule is of the following form:

$$r : \left(\biguplus_{i\in\mathcal{I}_n} [k_i]\,P_i\right) \setminus \left(\biguplus_{i\in\mathcal{I}_n} [k_i]\,S_i\right) \iff G \mid \exists \bar{x}.\ \left(\biguplus_{i\in\mathcal{I}_n} [k_i]\,D_i\right), \left(\biguplus_{j\in\mathcal{I}_m} [k_j]\,D_j\right), \left(\biguplus_{l\in\mathcal{I}_e} [k_l]\,D_l\right)$$

where for $i \in \mathcal{I}_n$, k_j are *matching locations*, and $j \in \mathcal{I}_m$ such that $k_j \in \left(\mathrm{FV}(\biguplus_{i\in\mathcal{I}_n} P_i) \cup \mathrm{FV}(\biguplus_{i\in\mathcal{I}_n} S_i)\right)$ are *non-matched forwarding locations*, and $l \in \mathcal{I}_e$ such that $k_l \in \bar{x}$ are *existential forwarding locations*.

The matching locations of the rule r are the set of locations k_i labeled by $i \in \mathcal{I}_n$. These are the locations that will be involved in rule matching and correspond to the set of locations $\mathrm{Locs}(\left(\biguplus_{i\in\mathcal{I}_n} [k_i]\,P_i, [k_i]\,S_i\right))$. We call each "$P_i, S_i$" belonging to a location k_i the *matching obligations* of k_i and assume that each location k_i has a non-empty matching obligation (i.e., either $P_i \neq \emptyset$ or $S_i \neq \emptyset$). All locations that appear in localization operators of the right-hand side of rule r are called forwarding locations. We classify them into three types: first we have *matched forwarding locations* in the rule body $\left(\biguplus_{i\in\mathcal{I}_n} [k_i]\,D_i\right)$ are such that each k_i is a matching location as well. Next, *non-matched forwarding locations* within $\left(\biguplus_{j\in\mathcal{I}_m} [k_j]\,D_j\right)$ are such that k_j is not a matching location, but appear as a term argument of some constraint in the rule head[5]. Finally, *existential forwarding locations* within $\left(\biguplus_{l\in\mathcal{I}_e} [k_l]\,D_l\right)$ are such that $k_l \in \bar{x}$, the set of existential variables. Note that this implicitly means that k_l neither is a matching location nor appear as a term argument of a rule head, hence it is a reference to a *new* location. We assume that the body of matched forwarding locations (i.e., D_i) may be empty but that of non-matched and existential forwarding locations must be non-empty (i.e., D_j and D_l). This assumption provide a more concise notation for n-neighbor restricted rules without superfluous references to locations that are not involved in the rule application[6]. We will refer to a n-neighbor restricted program as a general CHR^e program, or simply a CHR^e program. A well-formed CHR^e rule where heads are located at exactly one location are, by definition, 0-neighbor restricted. We call these *local rules*. A CHR^e program $\mathcal{P}$ is *n-neighbor restricted* if each rule in $\mathcal{P}$ are m-neighbor restricted for $m \leq n$.

As an example, consider the example program in Figure 1. Rules *base* and *elim* are examples of 0-neighbor restricted rules or local rules. This is because the each have rule heads that specify constraints from a single location. Rule *trans* however, has rule heads from two distinct locations X and Y. Specifically, rule heads $P = [X]\,edge(Y,D), [Y]\,path(Z,D')$ and $S = \emptyset$, satisfying the directly connected condition because we have $edge(Y,D)$ located at X, while satisfying the neighbor isolation condition by default because the rule only has one neighboring matching location Y. Hence it is a 1-neighbor restricted rule. An interested observation is that the 1-neighbor restriction corresponds directly to the link restriction of [2, 8].

$$r\_split : [X]\,unsorted(Xs) \iff \mathrm{len}(Xs) \geq 2 \mid \exists Y, Z.\ [Y]\,parent(X), [Y]\,unsorted(\mathrm{takeHalf}(Xs)), [Z]\,parent(X), [Z]\,unsorted(\mathrm{dropHalf}(Xs))$$

$$r\_base : [X]\,unsorted(Xs) \iff \mathrm{len}(Xs) < 2 \mid [X]\,sorted(Xs)$$

$$r\_ret : [Y]\,sorted(Xs), [Y]\,parent(X) \iff [X]\,unmerged(Xs)$$

$$r\_merge : [X]\,unmerged(Xs1), [X]\,unmerged(Xs2) \iff [X]\,sorted(\mathrm{merge}(Xs1, Xs2))$$

Figure 4. Parallel Mergesort in CHR^e

5. Parallel and Distributed Programming in CHR^e

In this section, we briefly discuss some practical examples of parallel and distributed programming in CHR^e. In the examples here, we will rely on term level functions to implement sequential subroutines (e.g., splitting a list, retrieve element, length of list, etc..) while higher level rewriting semantics of CHR rules concisely describes synchronization between locations. For clarity, we will write constraint predicate in *math* font, while term functions in normal font.

Figure 4 shows a parallel implementation of mergesort in CHR^e. We assume that we have four term functions defined as part of the built-in term language: namely $\mathrm{len}(Xs)$ that returns the length of list Xs, $\mathrm{takeHalf}(Xs)$ and $\mathrm{dropHalf}(Xs)$ that returns the first and second half of Xs, and $\mathrm{merge}(Xs1, Xs2)$ that merges $Xs1$ and $Xs2$ into a single sorted list[7]. Assume that, initially, we have the constraint $unsorted(Xs)$ at some location X, where Xs is the list to be sorted. For a non-empty and non-singleton Xs, the r_split rule splits a $unsorted(Xs)$ constraint into two halves ($unsorted(\mathrm{takeHalf}(Xs))$ and $unsorted(\mathrm{dropHalf}(Xs))$) and *creates* two new location Y and Z, each containing the respective halves. X will be designated as the parent of Y and Z, represented by the constraints $[Y]\,parent(X)$ and $[Z]\,parent(X)$. The r_base rule rewrites $unsorted(Xs)$ to $sorted(Xs)$ if Xs is a singleton or empty list, while r_ret states the rewriting of

[5] The *directly connected* condition of neighbor restriction dictates that the primary matching location would possess one such constraint.

[6] A matched forwarding location k_i is allowed to have an empty rule body D_i because its presence is justified by its appearance as a matching location.

[7] In essence, this is the style of distributed programming which we advocate, where term expressions describe sequential operations while synchronization and concurrency is handled at the multiset rewriting level.

r_local : $[X]\,unsorted(Xs) \iff [X]\,sorted(\mathrm{sort}(Xs))$

r_done : $[X]\,leader(), [X]\,leaderLinks([\_]) \iff true$

r_bcast : $[X]\,sorted(Xs), [X]\,leader() \setminus [X]\,leaderLinks(G) \iff$
$[X]\,leaderLinks(\mathrm{takeHalf}(G)),$
$[\mathrm{elemAt}(\mathrm{len}(Xs)/2, G)]\,leader(),$
$[\mathrm{elemAt}(\mathrm{len}(Xs)/2, G)]\,leaderLinks(\mathrm{dropHalf}(G)),$
$[X]\,bcastMedian(\mathrm{elemAt}(\mathrm{len}(Xs)/2, Xs), G)$
$[X]\,bcastPartners(\mathrm{takeHalf}(G), \mathrm{dropHalf}(G))$

$r\_bcastM1$: $[X]\,bcastMedian(\_, [\,]) \iff true$

$r\_bcastM2$: $[X]\,bcastMedian(M, Y : Ys) \iff$
$[Y]\,median(M), [X]\,bcastMedian(M, Ys)$

$r\_bcastP1$: $[X]\,bcastPartners([\,],[\,]) \iff true$

$r\_bcastP2$: $[X]\,bcastPartners(Y : Ys, Z : Zs) \iff$
$[Y]\,partnerLink(Z), [X]\,bcastPartners(Ys, Zs)$

r_part : $[X]\,median(M), [X]\,sorted(Xs) \iff$
$[X]\,leqM(\mathrm{filterLeq}(M, Xs)), [X]\,grM(\mathrm{filterGr}(M, Xs))$

r_swap : $[X]\,partnerLink(Y), [X]\,grM(Xs), [Y]\,leqM(Ys) \iff$
$[X]\,leqM(Ys), [Y]\,grM(Xs)$

r_leq : $[X]\,leqM(Ls1), [X]\,leqM(Ls2) \iff$
$[X]\,sorted(\mathrm{merge}(Ls1, Ls2))$

r_gr : $[X]\,grM(Gs1), [X]\,grM(Gs2) \iff$
$[X]\,sorted(\mathrm{merge}(Gs1, Gs2))$

Figure 5. Distributed Hyper Quicksort in CHR^e

$sorted(Xs)$ at location Y to $unmerged(Xs)$ at the parent of Y (indicated by $[Y]\,parent(X)$). Finally, r_merge rule states that constraints $merged(Xs1)$ and $merged(Xs2)$ located at the same location X is rewritten to $sorted(\mathrm{merge}(Xs1, Xs2))$.

Note this implementation is 'parallel' because we assume a tightly coupled concurrent execution: While each created location sorts a separate fragment of the list, the 'split' and 'merge' steps (r_split and r_merge rules) 'moves' each element of the original list Xs across $log\ n$ locations. Depending on the underlying architecture, new locations created by existential quantification (e.g., in r_split rule) may or may not be mapped to new physical network locations. For example, in a multicore system, they could be mapped to processes to be executed by a pool of local computing entities operating over a shared memory space.

We now consider another sorting algorithm which is more suitable for a distributed context. Figure 5 shows an implementation of distributed hyper quicksort [12]. Hyper quicksort is a 'distributed' sorting algorithm because it assumes that we have m lists of integers distributed between m locations to be globally sorted and that it is not practical to consolidate all lists into one centralized location. In addition to the four term functions introduced earlier, we assume we have the following: $\mathrm{sort}(Xs)$ sequentially sorts Xs, $\mathrm{elemAt}(I, Xs)$ returns the I^{th} element of Xs, $\mathrm{filterLeq}(M, Xs)$ and $\mathrm{filterGr}(M, Xs)$ returns list of all elements in Xs less than equal and greater than M respectively. Initially, each constraint store begins with an $unsorted(Xs)$ constraint (Xs contains the list unique to that location) and exactly one location is nominated the leader by having the additional constraints $leader()$ and $leaderLink(G)$ such that G contains the list of all location identifiers. The algorithm begins by locally sorting each list, using $\mathrm{sort}(Xs)$ called by the rule r_local. The rule r_done states that if we have $leaderLink(G)$ such that G is a singleton, sorting is done and we can remove the leader constraints. The rule r_bcast states that for a leader location X, given that we have $sorted(Xs)$ at X, we split all locations in G into two halves $Lg = \mathrm{takeHalf}(G)$ and $Gg = \mathrm{dropHalf}(G)$. X will remain the leader of Lg while first element of Gg is designated as the leader of Gg (i.e., $[\mathrm{elemAt}(\mathrm{len}(Xs)/2, G)]\,leader()$). Next, the median element of Xs ($M = \mathrm{elemAt}(\mathrm{len}(Xs)/2, Xs)$) is broadcast to each location in G (modeled by $[X]\,bcastMedian(M, G)$ and the $r\_bcastM1$ and $r\_bcastM2$ rules). Finally, each location in Lg is assigned a unique partner in Gg (modeled by $[X]\,bcastPartners(Lg, Gg)$ and the $r\_bcastP1$ and $r\_bcastP2$ rules). The rule r_part states that, given median M, we partition a sorted list Xs into elements less than M ($leqM(Ls)$) and elements greater than M ($grM(Gs)$). Rule r_swap states that a location X with $partnerLink(Y)$ swaps with Y its elements greater than M ($grM(Xs)$) for Y's elements less than M ($leqM(Ys)$). Rules r_leq and r_gr finally merges pairs of $leqM$ and grM constraints into one $sorted$ constraint.

All rules except for r_swap are 0-neighbor restricted, hence rewritings of such rules are local. Rule r_swap is 1-neighbor restricted, hence require synchronization between two locations. The purpose of assigning partners in the rule r_bcast (i.e., $[X]\,bcastPartners(Lg, Gg)$) is such that we can write r_swap as 1-neighbor restricted. Hence synchronization does not involve arbitral locations, rather a location P is bound to only synchronizing with a location Q (via neighbor restriction satisfied by $[P]\,partnerLink(Q)$).

An interesting observation is that in place of the r_local rule, we can instead insert the parallel mergesort rules of Figure 4 into the example of Figure 5. This makes the sorting within each location locally execute in "parallel" (suppose each initial location is a multicore processor and r_split rule "creates" new processes as oppose to new physical locations). This demonstrates how CHR^e can provide an abstraction that blends together parallel and distributed programming.

6. Semantics of CHR^e

We define the ω_α^e abstract semantics of CHR^e in Section 6.1, which introduces a decentralized execution of CHR^e programs, and prove its soundness with respect to the ω_α semantics. Using ω_α^e as a stepping stone, Section 6.2 defines and proves the soundness of the ω_0^e operational semantics that provides a more operational view of decentralized execution of CHR^e programs. Although this operational semantics only supports 0-neighbor restricted rules, Section 7 will show how we encode the arbitral CHR^e programs into 0-neighbor restricted programs. Note that in this work, we assume a lossless network, meaning that communication between locations are always eventually delivered and never lost.

6.1 ω_α^e Abstract Semantics

In this section, we introduce ω_α^e, an abstract decentralized semantics for CHR^e. This semantics accounts for the distributed nature of CHR^e, where each location has its own constraint store. This abstract semantics models a state transition system between abstract ensemble states, which are multisets of local stores $\langle \bar{\mathcal{S}} \rangle_k$ where $\bar{\mathcal{S}}$ is a constraint store and k a location name. An abstract ensemble states $\mathcal{A}$ is well-formed if all constraint stores $\bar{\mathcal{S}}$ that appear in it are well-formed and location names k are unique.

Figure 6 shows the ω_α^e semantics. Given a CHR^e program $\mathcal{P}$, we write a derivation step of ω_α^e as $\mathcal{P} \rhd \mathcal{A} \mapsto_{\omega_\alpha^e} \mathcal{A}'$ for abstract states $\mathcal{A}, \mathcal{A}'$. A derivation step defines the application of an n-neighbor restricted rules: Each of the $n+1$ locations k_i for $i \in \mathcal{I}_n$ provides a partial match P_i and S_i in their respective stores to their respective matching obligations (i.e., P_i' and S_i'), resulting in the combined substitution θ. If guard θG is valid, we apply the rule instance by removing S_i from the respective store of matching location k_i and replace them with the respective normalized rule body fragment (D_i of matching location k_i). For non-matching forwarding locations k_j for $j \in \mathcal{I}_m$, we simply add rule body fragment D_j to their stores. Finally, for existential forwarding locations, we create new location names k_l that contains just the

$$\text{Constraint Store} \quad \bar{\mathcal{S}} \quad ::= \quad \emptyset \mid \bar{\mathcal{S}}, c \qquad\qquad \text{Abstract Ensemble} \quad \mathcal{A} \quad ::= \quad \emptyset \mid \langle \bar{\mathcal{S}} \rangle_k, \mathcal{A}$$

$$\frac{\begin{array}{c} r : \left(\uplus_{i \in \mathcal{I}_n} [k'_i]\, P'_i\right) \setminus \left(\uplus_{i \in \mathcal{I}_n} [k'_i]\, S'_i\right) \iff G \mid \exists \bar{x}.\, \bar{D} \in \mathcal{P} \\ \models \theta G \quad \left(\uplus_{i \in \mathcal{I}_n} P_i\right) = \theta\left(\uplus_{i \in \mathcal{I}_n} P'_i\right) \quad \left(\uplus_{i \in \mathcal{I}_n} S_i\right) = \theta\left(\uplus_{i \in \mathcal{I}_n} S'_i\right) \quad \left(\uplus_{i \in \mathcal{I}_n \cup \mathcal{I}_m} k_i\right) = \theta\left(\uplus_{i \in \mathcal{I}_n \cup \mathcal{I}_m} k'_i\right) \\ \left(\uplus_{i \in \mathcal{I}_n} [k_i]\, D_i, \uplus_{j \in \mathcal{I}_m} [k_j]\, D_j, \uplus_{l \in \mathcal{I}_e} [k_l]\, D_l\right) = \mathrm{NF}(\mathrm{Inst}(\theta(\exists \bar{x}.\, \bar{D}))) \end{array}}{\mathcal{P} \rhd \mathcal{A}, \left(\begin{array}{l} \uplus_{i \in \mathcal{I}_n} \langle \bar{\mathcal{S}}_i, P_i, S_i \rangle_{k_i}, \\ \uplus_{j \in \mathcal{I}_m} \langle \bar{\mathcal{S}}_j \rangle_{k_j} \end{array}\right) \mapsto^e_{\omega_\alpha} \mathcal{A}, \left(\begin{array}{l} \uplus_{i \in \mathcal{I}_n} \langle \bar{\mathcal{S}}_i, P_i, D_i \rangle_{k_i}, \\ \uplus_{j \in \mathcal{I}_m} \langle \bar{\mathcal{S}}_j, D_j \rangle_{k_j}, \\ \uplus_{l \in \mathcal{I}_e} \langle D_l \rangle_{k_l} \end{array}\right)}$$

$$\text{where} \quad \begin{array}{ll} k'_j \text{ for } j \in \mathcal{I}_m \text{ such that } k'_j \in \left(\mathrm{FV}(\uplus_{i \in \mathcal{I}_n} P_i, \uplus_{i \in \mathcal{I}_n} S_i)\right) & k'_l \text{ for } l \in \mathcal{I}_e \text{ such that } k'_l \in \bar{x} \\ \bar{D} \quad = \left(\uplus_{i \in \mathcal{I}_n} [k'_i]\, D'_i\right), \left(\uplus_{j \in \mathcal{I}_m} [k'_j]\, D'_j\right), \left(\uplus_{l \in \mathcal{I}_e} [k'_l]\, D'_l\right) & \end{array}$$

Figure 6. ω^e_α Abstract Semantics of CHR^e

$$\begin{array}{rl}
\text{Ensembles} & [\![\mathcal{A}, \langle \bar{\mathcal{S}} \rangle_k]\!] \;=\; [\![\mathcal{A}]\!], [\![\bar{\mathcal{S}}]\!]^k \qquad [\![\emptyset]\!] \;=\; \emptyset \\
\text{CHR Program} & [\![R\, \mathcal{P}]\!] \;=\; [\![R]\!]\, [\![\mathcal{P}]\!] \qquad [\![\cdot]\!] \;=\; \cdot \\
\text{CHR Rule} & [\![r : P \setminus S \iff G \mid B]\!] \\
 & \qquad = r : [\![P]\!] \setminus [\![S]\!] \iff G \mid [\![B]\!] \\
\text{Constraints} & [\![p(\vec{t})]\!]^l \;=\; p(l, \vec{t}) \\
\text{Stores} & \begin{cases} [\![c, \bar{\mathcal{S}}]\!]^l & = \quad [\![c]\!]^l, [\![\bar{\mathcal{S}}]\!]^l \\ [\![\emptyset]\!]^l & = \quad \emptyset \end{cases} \\
\text{Body} & \begin{cases} [\![\exists \bar{x}.\, D]\!] & = \quad \exists \bar{x}. [\![D]\!] \\ [\![\, [l] c,\, D]\!] & = \quad [\![c]\!]^l, [\![D]\!] \\ [\![true]\!] & = \quad true \end{cases} \\
\text{Head} & \begin{cases} [\![\, [l] c,\, H]\!] & = \quad [\![c]\!]^l, [\![H]\!] \\ [\![\cdot]\!] & = \quad \cdot \end{cases}
\end{array}$$

Figure 7. CHR Interpretation of CHR^e

rule body fragments D_l[8]. Since we assume that rule guards G have no side effects, $\models \theta G$ is a global assertion. Note that 0-neighbor restriction is the special case where matching is localized ($\mathcal{I}_n$ is a singleton set). We denote the reflexive and transitive application of derivation steps by $\mathcal{P} \rhd \mathcal{A} \mapsto^*_{\omega^e_\alpha} \mathcal{A}'$. ω^e_α derivation steps preserve the well-formedness of states $\mathcal{A}$, provided that CHR^e programs are well-formed (proven in [6]).

From here on, we will implicitly assume the well-formedness of $\mathcal{P}$, $\mathcal{A}$ and $\mathcal{A}'$, when writing $\mathcal{P} \rhd \mathcal{A} \mapsto^*_{\omega^e_\alpha} \mathcal{A}'$.

We now relate the CHR^e abstract semantics ω^e_α to the CHR abstract semantics ω_α. Figure 7 defines a function $[\![-]\!]$ that inductively traverses the structure of a CHR^e syntactic construct and translates it into a CHR construct that represents its CHR interpretation. For a CHR^e rule, this function translates located constraints $[l]\, p(\vec{t})$ to standard CHR constraints by inserting location l as the first (leftmost) term argument of the predicate (i.e., $p(l, \vec{t})$). We call these *location interpreted CHR constraints*. An abstract ensemble state $\mathcal{A}$ is interpreted in CHR simply by collapsing all constraint stores in $\mathcal{A}$ into a single global constraint store, containing location interpreted CHR constraints. The translation function is defined so that given a well-formed CHR^e syntactic object o, $[\![o]\!]$ is a well-formed syntactic object of CHR.

[8] New location names are implicitly created by the instantiation of existential variables (i.e., $\mathrm{Inst}(-)$) applied to the rule body after substitution θ is applied.

Theorem 1 states the soundness of ω^e_α, namely the translation function $[\![-]\!]$ preserves deriviability.

THEOREM 1 (Soundness of ω^e_α). *Given a CHR^e program $\mathcal{P}$ and abstract states $\mathcal{A}$ and $\mathcal{A}'$, if $\mathcal{P} \rhd \mathcal{A} \mapsto^*_{\omega^e_\alpha} \mathcal{A}'$, then $[\![\mathcal{P}]\!] \rhd [\![\mathcal{A}]\!] \mapsto^*_{\omega_\alpha} [\![\mathcal{A}']\!]$.*

6.2 ω^e_0 Operational Semantics

This section introduces the operational semantics ω^e_0. Similarly to ω^e_α, this semantics specifies a distributed execution for CHR^e programs. Unlike ω^e_α, it is *operational* in that it describes the execution of CHR^e programs in a procedural manner with a clear and concise execution strategy for each location of the ensemble. As such, it comprises more derivation rules, each of which is dedicated to a specific sub-task of decentralized multiset rewriting. The ω^e_0 semantics is an extension of the refined CHR operational semantics [3], adapted to describe an execution model for decentralized and incremental multiset matching[9]. As such, it shares many meta constructs with the refined CHR operational semantics (e.g., rule head occurrence index, goals, numbered constraints, history). We refer the interested reader to [3] for a more detailed treatment of the refined operational semantics of CHR. The ω^e_0 semantics applies only to 0-neighbor restricted rules. While we could easily have included n-neighbor restricted rule execution as a derivation step of ω^e_0, doing so would not capture the operational challenges of synchronizing multiple locations during rule application. Instead we compile n-neighbor restricted rules for $n > 1$ into 0-neighbor restricted rules in Section 7.

Figure 8 defines the states of the ω^e_0 semantics. An ω^e_0 ensemble Ω is a set of tuples of the form $\langle \vec{\mathcal{U}} \;;\; \vec{\mathcal{G}} \;;\; \bar{\mathcal{S}} \;;\; \vec{\mathcal{H}} \rangle_k$ which represent the state of the computing entity at location k. The *buffer* $\vec{\mathcal{U}}$ is a sequence of the constraints that have been sent to location k. The *goals* $\vec{\mathcal{G}}$ is a sequence of the constraints c or active constraints $c\#d : i$. The *numbered store* $\bar{\mathcal{S}}$ is a multiset of numbered constraints $c\#d$. The index d serves as a reference link between $c\#d$ in the store and an active constraint $c\#d : i$ in the goals. The *history* $\vec{\mathcal{H}}$ is a set of indices where each element is a unique set of constraint ids $(\vec{D})$. Note the use of accents to explicitly indicate the nature of each collection of objects. Buffers is a novel extension of this semantics while goals, numbered store and history are artifacts of the refined operational semantics of CHR. Also, like the refined operational semantics of CHR, we assume that each rule head in a CHR program have a unique rule occurrence index i, representing

[9] It is incremental in that multiset matches are processed incrementally from new information (constraints). This is a property that is crucial for any effective execution model for distributed rule-based systems.

Stored constraint id d Rule Occurrence Index i

Goals	g	$::=$	$b \mid c \mid c\#d : i$	Goals	$\vec{\mathcal{G}}$	$::=$	$\emptyset \mid g, \vec{\mathcal{G}}$
Ids	$\vec{D}$	$::=$	$\emptyset \mid d, \vec{D}$	Numbered Store	$\bar{\mathcal{S}}$	$::=$	$\emptyset \mid \bar{\mathcal{S}}, c\#d$
Buffers	$\vec{\mathcal{U}}$	$::=$	$\emptyset \mid c, \vec{\mathcal{U}}$	History	$\bar{\mathcal{H}}$	$::=$	$\emptyset \mid \bar{\mathcal{H}}, (\vec{D})$

Operational Ensembles $\Omega ::= \emptyset \mid \Omega, \langle \vec{\mathcal{U}} \,;\, \vec{\mathcal{G}} \,;\, \bar{\mathcal{S}} \,;\, \bar{\mathcal{H}} \rangle_k$

Figure 8. ω_0^e Ensemble States

the sequence (typically textual order of appearance) in which the rule head is matched to an active constraint. We write occurrence index i as a subscript of the rule head (i.e., $[l]\, c_i$). A state Ω is well-formed if each $\langle \vec{\mathcal{U}} \,;\, \vec{\mathcal{G}} \,;\, \bar{\mathcal{S}} \,;\, \bar{\mathcal{H}} \rangle_k \in \Omega$ has a unique location name k, all objects in $\vec{\mathcal{U}}$, $\vec{\mathcal{G}}$ and $\bar{\mathcal{S}}$ are ground and all term expressions that appear in them are well-formed. Furthermore we require that the goals $\vec{\mathcal{G}}$ has at most one active constraint ($c\#d : i$) found at the head of $\vec{\mathcal{G}}$ with a corresponding $c\#d$ found in the store $\bar{\mathcal{S}}$. Ω is *initial* if all $\vec{\mathcal{U}} = \emptyset$, $\bar{\mathcal{S}} = \emptyset$ and $\bar{\mathcal{H}} = \emptyset$ and *terminal* if all $\vec{\mathcal{U}} = \emptyset$ and $\vec{\mathcal{G}} = \emptyset$. We extend the meta operation $\mathrm{Locs}(-)$ to the domain of operational states Ω, namely that $\mathrm{Locs}(\Omega)$ returns the set of all locations k that appear in Ω.

We define three meta operations that will be used in the ω_0^e semantics: Given a CHR numbered store $\bar{\mathcal{S}}$, $\mathrm{DropIds}(\bar{\mathcal{S}})$ returns the multiset of all constraints in $\bar{\mathcal{S}}$ without their numbered ids; $\mathrm{Ids}(\bar{\mathcal{S}})$ returns the set of all constraint ids that appear in $\bar{\mathcal{S}}$; given a CHR program $\mathcal{P}$, $\mathrm{OccIds}(\mathcal{P})$ returns the set of all rule head occurrence indices that appear in each rule $R \in \mathcal{P}$.

Figure 9 defines the ω_0^e operational semantics. Given a 0-neighbor restricted CHR^e program $\mathcal{P}$, a ω_0^e derivation step expresses a transition between ensemble states, written $\mathcal{P} \rhd \Omega \mapsto_{\omega_0^e} \Omega'$. Execution in a location $\langle \vec{\mathcal{U}} \,;\, \vec{\mathcal{G}} \,;\, \bar{\mathcal{S}} \,;\, \bar{\mathcal{H}} \rangle_k \in \Omega$ is mainly driven by the goals $\vec{\mathcal{G}}$ which function as a stack of procedures waiting to be executed. By contrast $\vec{\mathcal{U}}$ buffers the constraints sent to the location. The (Flush) step states that constraints in a non-empty buffer $\vec{\mathcal{U}}$ are to be moved into the goals if the current goal is empty. (Loc 1), (Loc 2) and (Loc 3) model the delivery of body constraint $[k']\, c$ to forwarding location k': (Loc 1) applies if the leading goal is $[k']\, c$ and k' is distinct from the origin location k, and sends c to the buffer $\vec{\mathcal{U}}'$ of k'. For (Loc 2) the forwarding location is the origin location, hence no actual transmission occur and the localization operator $[k]$ is simply stripped away. Finally for (Loc 3) with localization operator $[k']$, k' does not appear anywhere in the ensemble, hence we create a new location k' with just c in the buffer and all other collections empty[10]. (Act) applies to a leading goal of the form $p(\vec{t})$. It introduces $p(\vec{t}) : d$ into the store, where d is a fresh constraint identifier, and puts the active constraint $p(\vec{t})\#d : 1$ as the new leading goal. This represents the initialization of rule matching rooted at $p(\vec{t})\#d$ starting from the rule head in the program $\mathcal{P}$ that corresponds to the first occurrence index. The last four derivation steps apply to leading goals of the form $c\#d : i$. (Prop) and (Simp) model the application of a 0-neighbor restricted CHR rule instance $R \in \mathcal{P}$ such that the i^{th} rule head occurrence matches c and respective partner constraints are found in the store. (Prop) additionally enforces a history check similar to the traditional CHR semantics [3, 13]. The purpose of this is to disallow multiple applications of rule instances that originate from the same multiset of constraints in the store[11]. The derivation (Next) increments the occurrence index of the active constraint, while (Drop) removes the active constraint from the goal once it has been tried on all rule head occurrences. We define the transitive and reflexive application of ω_0^e derivation steps as $\mathcal{P} \rhd \Omega \mapsto^*_{\omega_0^e} \Omega'$. A state Ω' is *reachable* by a program $\mathcal{P}$ under the ω_0^e semantics if there exists some initial state Ω such that $\mathcal{P} \rhd \Omega \mapsto^*_{\omega_0^e} \Omega'$.

We are interested in a class of CHR^e programs known as *locally quiescent* CHR^e programs. We say that a CHR^e program $\mathcal{P}$ is locally quiescent if given any well-formed reachable state Ω, we cannot have any infinite derivation sequences that *does not include* the (Flush) derivation step. This specifically means that each location k in a Ω must always (eventually and asynchronously) execute to a state where its goals are empty, during which the (Flush) step is applicable and the constraints in the buffer will be pushed into the goals. Locally quiescent programs have the property that constraints sent across locations are not left to "starve" in a location's buffer, because local execution of a location k is guaranteed to be terminating. Hence it is a form of progress guarantee that each location will eventually process (activate, store and match) each constraint delivered to it. While we expect that it is reasonable that distributed applications can possibly behave in a non *globally quiescent* manner [12], we expect that each location executes in a locally quiescent manner as described here. We will explicitly consider locally quiescent programs in Section 7.

Given a state Ω, we define a meta operation $\mathrm{Goals}(\Omega)$ that denotes the consolidated sequence of all goals in Ω. We extend the notion of location retrieval and location restriction on goals: $\mathrm{Locs}(\vec{\mathcal{G}})$ denotes the set of distinct locations k such that $[k]\, c \in \vec{\mathcal{G}}$ for some c. Given location k, $\vec{\mathcal{G}}_{|k}$ denotes the sequence containing all constraints c where $[k]\, c \in \vec{\mathcal{G}}$.

Figure 10 gives the translation function $\lceil - \rceil$ that inductively traverses the structure of an operational ensemble state Ω and translates it to a corresponding fragment of an abstract state $\mathcal{A}$. At the top-most level, $\lceil \Omega \rceil$ is equal to $\lceil \Omega \rceil^{\vec{\mathcal{G}}'}, \uplus_{k \in (\mathrm{Locs}(\vec{\mathcal{G}}') - \mathrm{Locs}(\Omega))} \langle \vec{\mathcal{G}}'_{|k} \rangle_k$ such that $\vec{\mathcal{G}}' = \mathrm{Goals}(\Omega)$ is the consolidated sequence of all goals in the ensemble. The first component ($\lceil \Omega \rceil^{\vec{\mathcal{G}}'}$) takes each location in Ω and collapse $\vec{\mathcal{U}}$, $\vec{\mathcal{G}}$ and $\bar{\mathcal{S}}$ together while dropping the history $\bar{\mathcal{H}}$ entirely. Kept as a superscript is the sequence of all goals $\vec{\mathcal{G}}'$. Each location k also extracts from $\vec{\mathcal{G}}'$ all located constraints that belong to k (i.e., $\vec{\mathcal{G}}'_{|k}$). Buffer $\vec{\mathcal{U}}$ is interpreted as a multiset of constraints. For $\vec{\mathcal{G}}$, we discard active constraints $c\#d : i$ because well-formed states have a corresponding $c\#d$ in $\bar{\mathcal{S}}$. We also discard located constraints $[k]\, c$. For $\bar{\mathcal{S}}$, we strip away constraint ids $\#d$. We implicitly convert sequences ($\vec{\mathcal{U}}$ and $\vec{\mathcal{G}}$) into multisets. The second component $\uplus_{k \in (\mathrm{Locs}(\vec{\mathcal{G}}') - \mathrm{Locs}(\Omega))} \langle \vec{\mathcal{G}}'_{|k} \rangle_k$ retrieves the set

[10] New locations are introduced by existential forwarding locations in rule bodies.

[11] History checking is only required on propagation rules ($r : P \setminus S \iff G \mid B$ where $S = \emptyset$). For brevity, we conservatively apply history checking to all states that applies to the (Prop) derivation step.

[12] Globally quiescent: collective execution of the ensemble terminates. In other words, there exists a reachable state where all locations of the ensemble reaches quiescence.

(Flush)
$$\frac{\vec{\mathcal{U}} \neq \emptyset}{\mathcal{P} \rhd \Omega, \langle \vec{\mathcal{U}} \,;\, \emptyset \,;\, \bar{\mathcal{S}} \,;\, \bar{\mathcal{H}} \rangle_k \;\mapsto_{\omega_0^e}\; \Omega, \langle \emptyset \,;\, \vec{\mathcal{U}} \,;\, \bar{\mathcal{S}} \,;\, \bar{\mathcal{H}} \rangle_k}$$

(Loc 1)
$$\frac{}{\mathcal{P} \rhd \Omega, \left(\begin{array}{l} \langle \vec{\mathcal{U}} \,;\, ([k']c, \vec{\mathcal{G}}) \,;\, \bar{\mathcal{S}} \,;\, \bar{\mathcal{H}} \rangle_k, \\ \langle \vec{\mathcal{U}}' \,;\, \vec{\mathcal{G}}' \,;\, \bar{\mathcal{S}}' \,;\, \bar{\mathcal{H}}' \rangle_{k'} \end{array} \right) \mapsto_{\omega_0^e} \Omega, \left(\begin{array}{l} \langle \vec{\mathcal{U}} \,;\, \vec{\mathcal{G}} \,;\, \bar{\mathcal{S}} \,;\, \bar{\mathcal{H}} \rangle_k, \\ \langle (\vec{\mathcal{U}}', [c]) \,;\, \vec{\mathcal{G}}' \,;\, \bar{\mathcal{S}}' \,;\, \bar{\mathcal{H}}' \rangle_{k'} \end{array} \right)}$$

(Loc 2)
$$\frac{}{\mathcal{P} \rhd \Omega, \langle \vec{\mathcal{U}} \,;\, ([k]c, \vec{\mathcal{G}}) \,;\, \bar{\mathcal{S}} \,;\, \bar{\mathcal{H}} \rangle_k \;\mapsto_{\omega_0^e}\; \Omega, \langle \vec{\mathcal{U}} \,;\, (c, \vec{\mathcal{G}}) \,;\, \bar{\mathcal{S}} \,;\, \bar{\mathcal{H}} \rangle_k}$$

(Loc 3)
$$\frac{k \neq k' \qquad k' \notin \mathrm{Locs}(\Omega)}{\mathcal{P} \rhd \Omega, \langle \vec{\mathcal{U}} \,;\, ([k']c, \vec{\mathcal{G}}) \,;\, \bar{\mathcal{S}} \,;\, \bar{\mathcal{H}} \rangle_k \;\mapsto_{\omega_0^e}\; \Omega, \langle \vec{\mathcal{U}} \,;\, \vec{\mathcal{G}} \,;\, \bar{\mathcal{S}} \,;\, \bar{\mathcal{H}} \rangle_k, \langle c \,;\, \emptyset \,;\, \emptyset \,;\, \emptyset \rangle_{k'}}$$

(Act)
$$\frac{d \text{ is a fresh id}}{\mathcal{P} \rhd \Omega, \langle \vec{\mathcal{U}} \,;\, (p(\vec{t}), \vec{\mathcal{G}}) \,;\, \bar{\mathcal{S}} \,;\, \bar{\mathcal{H}} \rangle_k \;\mapsto_{\omega_0^e}\; \Omega, \langle \vec{\mathcal{U}} \,;\, (p(\vec{t})\#d : 1, \vec{\mathcal{G}}) \,;\, (\bar{\mathcal{S}}, p(\vec{t})\#d) \,;\, \bar{\mathcal{H}} \rangle_k}$$

(Simp)
$$\frac{\begin{array}{c} r : [l]\,P' \setminus [l]\,(S', c_i', S'') \Longleftrightarrow G \;\mid\; B \in \mathcal{P} \qquad \models \theta \wedge G \qquad k = \theta l \\ \mathrm{DropIds}(P) = \theta P' \qquad \mathrm{DropIds}(S) = \theta(S', S'') \qquad c = \theta c' \end{array}}{\mathcal{P} \rhd \Omega, \langle \vec{\mathcal{U}} \,;\, (c\#d : i, \vec{\mathcal{G}}) \,;\, (\bar{\mathcal{S}}, P, S, c\#d) \,;\, \bar{\mathcal{H}} \rangle_k \;\mapsto_{\omega_0^e}\; \Omega, \langle \vec{\mathcal{U}} \,;\, (\mathrm{NF}(\mathrm{Inst}(\theta B)), \vec{\mathcal{G}}) \,;\, (\bar{\mathcal{S}}, P) \,;\, \bar{\mathcal{H}} \rangle_k}$$

(Prop)
$$\frac{\begin{array}{c} r : [l]\,(P', c_i', P'') \setminus [l]\,S' \Longleftrightarrow G \;\mid\; B \in \mathcal{P} \qquad \models \theta \wedge G \qquad k = \theta l \\ \mathrm{DropIds}(P) = \theta(P', P'') \qquad \mathrm{DropIds}(S) = \theta S' \qquad c = \theta c' \qquad (d, \mathrm{Ids}(P, S)) \notin \bar{\mathcal{H}} \end{array}}{\begin{array}{ll} \mathcal{P} \rhd & \Omega, \langle \vec{\mathcal{U}} \,;\, (c\#d : i, \vec{\mathcal{G}}) \,;\, (\bar{\mathcal{S}}, P, S, c\#d) \,;\, \bar{\mathcal{H}} \rangle_k \\ \mapsto_{\omega_0^e} & \Omega, \langle \vec{\mathcal{U}} \,;\, (\mathrm{NF}(\mathrm{Inst}(\theta B)), c\#d : i, \vec{\mathcal{G}}) \,;\, (\bar{\mathcal{S}}, P, c\#d) \,;\, (\bar{\mathcal{H}}, (d, \mathrm{Ids}(P, S)))\rangle_k \end{array}}$$

(Next)
$$\frac{\text{(Simp) and (Prop) do not apply for } c\#d : i}{\mathcal{P} \rhd \Omega, \langle \vec{\mathcal{U}} \,;\, (c\#d : i, \vec{\mathcal{G}}) \,;\, \bar{\mathcal{S}} \,;\, \bar{\mathcal{H}} \rangle_k \;\mapsto_{\omega_0^e}\; \Omega, \langle \vec{\mathcal{U}} \,;\, (c\#d : (i+1), \vec{\mathcal{G}}) \,;\, \bar{\mathcal{S}} \,;\, \bar{\mathcal{H}} \rangle_k}$$

(Drop)
$$\frac{i \notin \mathrm{OccIds}(\mathcal{P})}{\mathcal{P} \rhd \Omega, \langle \vec{\mathcal{U}} \,;\, (c\#d : i, \vec{\mathcal{G}}) \,;\, \bar{\mathcal{S}} \,;\, \bar{\mathcal{H}} \rangle_k \;\mapsto_{\omega_0^e}\; \Omega, \langle \vec{\mathcal{U}} \,;\, \vec{\mathcal{G}} \,;\, \bar{\mathcal{S}} \,;\, \bar{\mathcal{H}} \rangle_k}$$

Figure 9. ω_0^e Operational Semantics for CHR^e

$$\text{Ensembles} \quad \begin{cases} \lceil \Omega \rceil = \lceil \Omega \rceil^{\vec{\mathcal{G}}'}, \biguplus_{k \in (\mathrm{Locs}(\vec{\mathcal{G}}') - \mathrm{Locs}(\Omega))} \langle \vec{\mathcal{G}}'_{|k} \rangle_k \\ \qquad \text{where } \vec{\mathcal{G}}' = \mathrm{Goals}(\Omega) \\ \lceil \Omega, \langle \vec{\mathcal{U}} \,;\, \vec{\mathcal{G}} \,;\, \bar{\mathcal{S}} \,;\, \bar{\mathcal{H}} \rangle_k \rceil^{\vec{\mathcal{G}}'} \\ \qquad = \lceil \Omega \rceil^{\vec{\mathcal{G}}'}, \langle \lceil \vec{\mathcal{U}} \rceil, \lceil \vec{\mathcal{G}} \rceil, \lceil \bar{\mathcal{S}} \rceil, \vec{\mathcal{G}}'_{|k} \rangle_k \\ \lceil \emptyset \rceil^{\vec{\mathcal{G}}'} \;=\; \emptyset \end{cases}$$

$$\text{Identity} \quad \lceil \emptyset \rceil \;=\; \emptyset$$

$$\text{Buffers} \quad \lceil c, \vec{\mathcal{U}} \rceil \;=\; c, \lceil \vec{\mathcal{U}} \rceil$$

$$\text{Stores} \quad \lceil c\#d, \bar{\mathcal{S}} \rceil \;=\; c, \lceil \bar{\mathcal{S}} \rceil$$

$$\text{Goals} \quad \begin{cases} \lceil c, \vec{\mathcal{G}} \rceil & = & c, \lceil \vec{\mathcal{G}} \rceil \\ \lceil c\#d : i, \vec{\mathcal{G}} \rceil & = & \lceil \vec{\mathcal{G}} \rceil \\ \lceil [k]c, \vec{\mathcal{G}} \rceil & = & \lceil \vec{\mathcal{G}} \rceil \end{cases}$$

Figure 10. Abstract Ensemble State Interpretation of Operational Ensemble States

of all locations k that are referenced in the goals but are not known locations in Ω, and "creates" these new locations each containing their respective fragment of the goals (i.e., $\vec{\mathcal{G}}'_{|k}$).

Theorem 2 states that ω_0^e derivations can be mapped to corresponding ω_α^e derivations via the $\lceil - \rceil$ interpretation of operational states.

THEOREM 2 (Soundness of ω_0^e). *Given* 0*-neighbor restricted CHR*^{e} *program* $\mathcal{P}$ *and states* Ω *and* Ω'*, if* $\mathcal{P} \rhd \Omega \;\mapsto_{\omega_0^e}^*\; \Omega'$*, then* $\mathcal{P} \rhd \lceil \Omega \rceil \mapsto_{\omega_\alpha^e}^* \lceil \Omega' \rceil$.

Note that a terminal state Ω (where all $\vec{\mathcal{U}}_i = \emptyset$ and $\vec{\mathcal{G}}_i = \emptyset$) is in a state of *quiescence* where no ω_0^e derivation steps can apply. The ω_0^e semantics guarantees that when we reach quiescence from a well-formed initial state, all rules in the program have been exhaustively applied. Serializability of concurrent execution of ω_0^e derivations can be proven in manner similar to [4, 7]. Details of the proof of exhaustiveness and concurrency is found in the technical report [6].

7. Encoding 1-Neighbor Restriction

In this section, we define a translation that transforms a 1-neighbor restricted program into a 0-neighbor restricted program. Given a 1-neighbor restricted rule $r : [X]\,P_x\,,[Y]\,P_y \setminus [X]\,S_x\,,[Y]\,S_y \Longleftrightarrow G \mid B$, we designate X as the primary location and Y as the neighbor location. We define two properties of a 1-neighbor restricted rule r, namely that r is *primary propagated* if $S_x = \emptyset$ and that r is *neighbor propagated* if $S_y = \emptyset$. We call P_x and S_x the *primary matching obligations*, while P_y and S_y are the *neighbor matching obligations*. We only consider locally quiescent CHR^e programs from this section.

7.1 Basic Encoding Scheme

Figure 11 illustrates our basic encoding scheme, $\rightsquigarrow_{1\mathrm{Nb}}^{\mathrm{basic}}$ on an example. It applies to a 1-neighbor restricted rule that is neither

$$swap : \left(\begin{array}{c} [X]\, neighbor(Y), \\ \cdot \end{array} \right) \setminus \left(\begin{array}{c} [X]\, color(C), \\ [Y]\, color(C') \end{array} \right) \Longleftrightarrow \left(\begin{array}{c} [X]\, color(C'), \\ [Y]\, color(C) \end{array} \right)$$

$$\rightsquigarrow_{1\mathrm{Nb}}^{\mathrm{basic}} \left(\begin{array}{l} swap\_1 : [X]\, neighbor(Y), [X]\, color(C) \Longrightarrow [Y]\, swap\_req(X,Y,C) \\ swap\_2 : [Y]\, color(C') \setminus [Y]\, swap\_req(X,Y,C) \Longleftrightarrow [X]\, swap\_match(X,Y,C,C') \\ swap\_3 : [X]\, neighbor(Y) \setminus [X]\, color(C), [X]\, swap\_match(X,Y,C,C') \\ \qquad\qquad \Longleftrightarrow [Y]\, swap\_commit(X,Y,C,C') \\ swap\_4a : [Y]\, swap\_commit(X,Y,C,C'), [Y]\, color(C') \Longleftrightarrow [X]\, color(C'), [Y]\, color(C) \\ swap\_4b : [Y]\, swap\_commit(X,Y,C,C') \Longleftrightarrow [X]\, color(C) \end{array} \right)$$

Figure 11. Color Swapping Example: 1-Neighbor Restricted Rule

primary nor neighbor propagated. We assume that constraints of the predicate $neighbor$ are never deleted, i.e., no other rules in a program with the $swap$ rule have a $neighbor$ constraint as a simplified head. We call such a constraint a *persistent* constraint. A constraint c is *persistent* if there is no substitution θ such that $\theta c = \theta c'$ for some rule $r : P \setminus S \Longleftrightarrow G \mid B \in \mathcal{P}$ and constraint $c' \in S$. Note that the rule heads demand that we must *atomically observe* that in some location X we have $neighbor(Y)$ and $color(C)$, while in Y we have $color(C')$. This observation must be *atomic* in the sense that the observation of X's and Y's matching obligations should not be interrupted by an interleaving concurrent derivation.

The 0-neighbor restricted encoding of the $swap$ rule (Figure 11) recovers this atomicity: In $swap\_1$ X sends a swap request $swap\_req(X,Y,C)$ to Y if it possesses the primary matching obligation of the $swap$ rule. In $swap\_2$ if Y observes this request together with a $color(C')$, it responds to X by sending $swap\_match(X,Y,C,C')$. Note that in this example, $swap\_req(X,Y,C)$ is simplified since X's matching obligation is not primary propagated[13]. In $swap\_3$, X must observe that it has a response from Y ($swap\_match(X,Y,C,C')$) and that its matching obligations are still valid[14], then it sends a commit request to Y $swap\_commit(X,Y,C,C')$. This is the point where X actually *commits* to the match by consuming $color(C)$. From here there are two possibilities: $swap\_4a$ considers that, if Y still possesses the matching instance of $color(C')$, we complete the execution of $swap$ by delivering its rule body. $swap\_4b$ considers the alternative case where Y no longer has $color(C')$ hence we cannot commit to the rule instance. Hence we *roll back* X's commitment by returning $color(C)$ to X. Note that ω_0^e's sequencing of rule occurrences is vital to ensure that given a $swap\_commit$ instance, rule matching for $swap\_4a$ is always attempted before $swap\_4b$. We call the predicates introduced in the encoding (like $swap\_req$, $swap\_match$) *synchronizing predicates* and constraints they form *synchronizing constraints*.

The five 0-neighbor restricted rules in Figure 11 implement an *asynchronous* and *optimistic* synchronization protocol between two locations of the ensemble. It is asynchronous because neither primary X nor neighbor Y ever "blocks" or busy-waits for responses. Rather they communicate asynchronously via the synchronizing constraints, while potentially interleaving with other derivation steps. It is optimistic because non-synchronizing constraints are only ever consumed after both X and Y have independently observed their respective fragment of the rule head instance.

Figure 12 defines another example of a 1-neighbor restricted rule that differs from the $swap$ example in two ways: It is *primary propagated* and it contain a primary propagated head (namely $color(C)$) that is not *persistent*. We highlight in boxes the fragments which differ by the fact that the rule is *primary propagated*, and with an underline the fragments which differ by the fact that propagated head $color(C)$ is not *persistent*.

Since $prop$ is primary propagated, it is possible that a single instance of this rule head fragment be applied to multiple instances of $[Y]\, color(C')$ for a particular location Y. If we follow a similar encoding to the $swap\_2$ rule for the previous example (Figure 11) we cannot guarantee exhaustiveness of 1-neighbor restricted rule application. Therefore, as highlighted in a box in figure 12, rule $prop\_2$ is defined such that the synchronizing constraint $prop\_req(X,Y,C)$ is propagated as opposed to being simplified (highlighted in a box).

Since $prop$ has a non persistent propagated head, in order to safely guarantee that the observation of X and Y matching obligations are done independently, $prop3$ commits its obligation by deleting its non persistent propagated constraint(s), in this case $color(C)$, while in $prop4a$ and $prop4b$, this constraint is returned to X. While this possibly introduces additional overhead, it is crucial to ensuring the safety of this rule application.

Figure 13 defines the basic encoding scheme for 1-neighbor restricted rule. It is denoted by $R_1 \rightsquigarrow_{1\mathrm{Nb}}^{\mathrm{basic}} \mathcal{P}_0$, where R_1 is a well-formed 1-neighbor restricted rule while $\mathcal{P}_0$ a well-formed 0-neighbor restricted program. The propagated rule heads of the primary location X are split into two, namely P_x containing all rule heads which are persistent in $\mathcal{P}_1$ and P'_x containing all those which are non-persistent. P'_x and S_x will be consumed in r_3 when the primary location commits, this effectively "locks" the primary matching obligation. The neighbor location Y applies r_4a to complete the rule application r, specifically adding the rule body D and "unlocking" P'_x, the primary propagated rule heads which are not persistent. If it is unable to complete this rule application, r_4b is applied to roll-back location X's commit attempt by returning P'_x and S_x to X. $\mathrm{MatchRule}(\cdot)$ characterizes the fragment of the encoding unique to primary propagated 1-neighbor restricted rules, while $\mathrm{MatchRule}(S_x)$ for a non-empty S_x represents the corresponding fragment of non-primary propagated rules. Xs and Ys are the set of variables of the primary and neighbor location rule heads while Rs is the union of the two. Rule guards are divided into two parts, namely G_x the primary rule guards and G_y the neighbor rule guards. Primary rule guards G_x are all the guard conditions that are grounded by Xs, while G_y are the rest of the guards. We refer to these rules (r_i) as *encoding rules*. The meta operator $\mathrm{DropSyncs}(\bar{\mathcal{S}}; \mathcal{P})$ denotes the multiset of all constraints that appear in $\bar{\mathcal{S}}$ that are not synchronizing constraints of $\mathcal{P}$.

[13] If it was primary propagated (by making $[X]\, color(C)$ of the $swap$ rule propagated instead), we would have to propagate $swap\_req(X,Y,C)$ instead, since X's matching obligation can match and apply to multiple instances of $swap$ (see Figure 12 for such an example).

[14] This revalidation ensures that Y's observation of its matching obligation has not been invalidated by an interleaving rule application that consumed any part of X's obligations.

$$prop : \left(\begin{array}{c} [X]\, neighbor(Y), \underline{[X]\, color(C)} \\ . \end{array} \right) \setminus \left(\begin{array}{c} \boxed{\cdot} \\ \underline{[Y]\, color(C')} \end{array} \right) \Longleftrightarrow [Y]\, color(C)$$

$$\rightsquigarrow^{\text{basic}}_{\text{1Nb}} \left(\begin{array}{l} prop1 : [X]\, neighbor(Y), [X]\, color(C) \Longrightarrow [Y]\, prop\_req(X,Y,C) \\ prop2 : \boxed{[Y]\, color(C'), [Y]\, prop\_req(X,Y,C) \Longrightarrow} [X]\, prop\_match(X,Y,C,C') \\ prop3 : [X]\, neighbor(Y) \setminus \underline{[X]\, color(C)}, [X]\, prop\_match(X,Y,C,C') \\ \qquad\qquad \Longleftrightarrow \underline{[Y]\, prop\_commit(X,Y,C,C')} \\ prop4a : [Y]\, prop\_commit(X,Y,C,C'), [Y]\, color(C') \Longleftrightarrow \underline{[X]\, color(C)}, [Y]\, color(C) \\ prop4b : [Y]\, prop\_commit(X,Y,C,C') \Longleftrightarrow \underline{[X]\, color(C)} \end{array} \right)$$

Figure 12. Color Propagation Example: Primary Propagated 1-Neighbor Restricted Rule

$$(r : [X]\, P_x, [X]\, P'_x, [Y]\, P_y \setminus [X]\, S_x, [Y]\, S_y \Longleftrightarrow G_x, G_y \mid \exists \bar{z}.\ D)$$

$$\rightsquigarrow^{\text{basic}}_{\text{1Nb}} \left(\begin{array}{l} r\_1 : [X]\, P_x, [X]\, S_x \Longrightarrow G_x \mid [Y]\, r\_req(Xs) \\ \text{MatchRule}(S_x) \\ r\_3 : [X]\, P_x \setminus [X]\, P'_x, [X]\, S_x, [X]\, r\_match(Rs) \Longleftrightarrow [Y]\, r\_commit(Rs) \\ r\_4a : [Y]\, P_y \setminus [Y]\, S_y, [Y]\, r\_commit(Rs) \Longleftrightarrow \exists \bar{z}.\ [X]\, P'_x, D \\ r\_4b : [Y]\, r\_commit(Rs) \Longleftrightarrow [X]\, P'_x, [X]\, S_x \end{array} \right)$$

where
All $c \in P_x$ are persistent constraints and all $c' \in P'_x$ are non-persistent constraints
$\text{MatchRule}(.) = r\_2 : [Y]\, P_y, [Y]\, S_y, [Y]\, r\_req(Xs) \Longrightarrow G_y \mid [X]\, r\_match(Rs)$
$\text{MatchRule}(S_x) = r\_2 : [Y]\, P_y, [Y]\, S_y \setminus [Y]\, r\_req(Xs) \Longleftrightarrow G_y \mid [X]\, r\_match(Rs) \quad \text{if } S_x \neq .$
$Xs = \text{FV}(P_x, P'_x, S_x) \qquad Ys = \text{FV}(P_y, S_y) \qquad Rs = \text{FV}(P_x, P'_x, S_x, P_y, S_y)$
$\text{FV}(G_x) \subseteq \text{FV}(P_x, P'_x, S_x) \qquad \text{FV}(G_y) \subseteq \text{FV}(P_x, P'_x, S_x, P_y, S_y)$

Figure 13. Basic Encoding of 1-Neighbor Restricted Rules

As illustrated in Figure 13, we generalize the application of this translation to 1-neighbor restricted *programs*, thus given a 1-neighbor restricted program $\mathcal{P}_1$, we have its encoding via $\mathcal{P}_1 \rightsquigarrow^{\text{basic}}_{\text{1Nb}} \mathcal{P}_0$, such that the encoding operation is applied to each 1-neighbor restricted rule in $\mathcal{P}_1$ while 0-neighbor restricted rules are simply left unmodified. All rule encodings (each a 0-neighbor program) are then concatenated into a single 0-neighbor restricted program $\mathcal{P}_0$. We assume that unique rule head occurrence indices are issued in order of rule head appearance in $\mathcal{P}_0$. When required for specific discussions, we will denote a (Simp) or (Prop) derivation step that involves the application of a r_i encoding rule instance as $\mathcal{P} \rhd \Omega \mapsto^{r\_i}_{\omega^e_0} \Omega'$, where Ω and Ω' are the states before and after the application of r_i.

It is possible that a partial sequences of encoding rules (r_i) is executed in a ω^e_0 derivation. For instance, primary location X can apply r_1 but never receives a reply from neighbor location Y with r_2 because Y does not possess the matching obligations required to complete the rule instance. Or similarly, Y can apply r_2 in response to X's instance of r_1, but never receives a reply from X with r_3 because X no longer possess matching obligations. Such partial sequences of execution are the side-effect of asynchrony in this synchronization protocol, and are *benign* in that they do not rewrite (delete or insert) non-synchronizing constraints. Furthermore, their only observable effects are the introduction of synchronizing constraints r_req and r_match whose only purpose and effect is the sequencing and staging of the flow of consensus building between locations X and Y.

We now consider the soundness of this encoding. Specifically, the soundness condition that we need is that the ω^e_0 derivations of the 0-neighbor restricted encodings of a 1-neighbor restricted program $\mathcal{P}$ derive valid states computable by $\mathcal{P}$ in the ω^e_α semantics. However, not all states derived by our encodings are such valid states: It is possible that the ω^e_0 derivations of the 0-neighbor restricted encodings derive intermediate states in which a 1-neighbor restricted rule instance is partially applied. Specifically, after application of r_3 only the X matching obligation is consumed, hence the rule instance is only partially applied. For this reason, the encodings are defined such that these intermediate states always contain the r_commit synchronizing constraint. We call such states *non-commit free* states and *commit free* states are all other states that do not contain r_commit. A state Ω is *commit free* if and only if for all $\langle \bar{\mathcal{S}} \rangle_k \in \lceil \Omega \rceil$, all $p(\vec{t}) \in \bar{\mathcal{S}}$ is such that $p \neq r\_commit$. Note we use $\lceil \Omega \rceil$ for the convenience of collapsing the contents of buffers, goals and stores of the ω^e_0 semantics into one abstract store of the ω^e_α semantics. By definition of the $\lceil - \rceil$ translation, this effectively means that all buffers, goals and store in the operational state Ω must not contain any commit synchronizing constraints in order for Ω to qualify as a commit-free state. An important property of this encoding is that non-commit free states can always be eventually returned to a commit free state by applying either r_4a or r_4b, resulting to the complete execution of r or a roll-back to the state before its execution attempt, respectively. Note that this is only true for CHR^e programs which are locally quiescent: The reason is because if a location's execution is not locally quiescent, it might never push buffered constraints into the goals (via the (Flush) derivation step of ω^e_0 semantics) and hence might remain non-commit free indefinitely.

Lemma 3 states that the encoding operation $\rightsquigarrow^{\text{basic}}_{\text{1Nb}}$ preserves local quiescence of CHR^e programs.

LEMMA 3 ($\rightsquigarrow^{\text{basic}}_{\text{1Nb}}$ Preserves Local Quiescence). *Given a locally quiescent 1-neighbor restricted program $\mathcal{P}_1$ and a 0-neighbor restricted program $\mathcal{P}_0$ such that $\mathcal{P}_1 \rightsquigarrow^{\text{basic}}_{\text{1Nb}} \mathcal{P}_0$, then $\mathcal{P}_0$ is also locally quiescent.*

$$blend : \left(\begin{array}{l} [X]\, neighbor(Y), \\ [Y]\, pallete(C') \end{array} \right) \setminus \left(\begin{array}{l} [X]\, color(C) \\ \boxed{\cdot} \end{array} \right) \Longleftrightarrow [Y]\, color(C + C')$$

$$\rightsquigarrow_{1\mathrm{Nb}}^{\mathrm{n-persist}} \left(\begin{array}{l} blend1 : [X]\, neighbor(Y), [X]\, color(C) \Longrightarrow [Y]\, blend\_req(X, Y, C) \\ blend2 : [Y]\, pallete(C') \setminus [Y]\, blend\_req(X, Y, C) \Longleftrightarrow [X]\, blend\_match(X, Y, C, C') \\ blend3 : [X]\, neighbor(Y) \setminus [X]\, color(C), [X]\, blend\_match(X, Y, C, C') \Longleftrightarrow [X]\, color(C + C') \end{array} \right)$$

$$(r : [X]\, P_x, [Y]\, P_y \setminus [X]\, S_x \Longleftrightarrow G_x, G_y \mid B) \rightsquigarrow_{1\mathrm{Nb}}^{\mathrm{n-persist}} \left(\begin{array}{l} r\_1 : [X]\, P_x, [X]\, S_x \Longrightarrow G_x \mid [Y]\, r\_req(Xs) \\ \mathrm{MatchRule}(S_x) \\ r\_3 : [X]\, P_x \setminus [X]\, S_x, [X]\, r\_match(Rs) \Longleftrightarrow B \end{array} \right)$$

where
All $c \in P_y$ are persistent constraints
$\mathrm{MatchRule}(.) = r\_2 : [Y]\, P_y, [Y]\, r\_req(Xs) \Longrightarrow G_y \mid [X]\, r\_match(Rs)$
$\mathrm{MatchRule}(S_x) = r\_2 : [Y]\, P_y \setminus [Y]\, r\_req(Xs) \Longleftrightarrow G_y \mid [X]\, r\_match(Rs) \quad \text{if } S_x \neq .$
$Xs = \mathrm{FV}(P_x, S_x)$ and $\mathrm{FV}(G_x) \subseteq \mathrm{FV}(P_x, S_x)$ and $\mathrm{FV}(G_y) \subseteq \mathrm{FV}(P_x, P_y, S_y)$

Figure 14. Optimized Encoding for Neighbor Persistent Rule

Lemma 4 states the property that non-commit free states can always eventually derive commit free state.

LEMMA 4 (1-Neighbor Commit-Free Reachability). *Given a locally quiescent* 1*-neighbor restricted program* $\mathcal{P}_1$ *and a* 0*-neighbor restricted program* $\mathcal{P}_0$ *such that* $\mathcal{P}_1 \rightsquigarrow_{1\mathrm{Nb}}^{\mathrm{basic}} \mathcal{P}_0$ *and states* Ω *reachable by* $\mathcal{P}_0$*, if* Ω *is not commit-free, then there exists some commit-free state* Ω' *such that* $\mathcal{P}_0 \rhd \Omega \mapsto_{\omega_0^e}^* \Omega'$.

Lemma 5 states that given any ω_α^e derivation between two commit free state $\mathcal{P}_0 \rhd \mathcal{A} \mapsto_{\omega_\alpha^e}^* \mathcal{A}'$, we can safely permute it such that all applications of r_3 encoding rules are immediately followed by either r_4a or r_4b. This lemma is proven by using the monotonicity property of the ω_α^e semantics (Details in [6]).

LEMMA 5 (Basic Encoding Rule Serializability). *Given a* 1*-neighbor restricted and locally quiescent CHR*^{e} *program* $\mathcal{P}_1$ *and a* 0*-neighbor restricted CHR*^{e} *program* $\mathcal{P}_0$ *such that* $\mathcal{P}_1 \rightsquigarrow_{1\mathrm{Nb}} \mathcal{P}_0$ *and commit free abstract states* $\mathcal{A}_1$ $\mathcal{A}_2$*,* $\mathcal{A}_3$ *and* $\mathcal{A}_4$*, given that:*

1. *For encoding rule instances* r_3 *and* r_4a*, we have* $\mathcal{P}_0 \rhd \mathcal{A}_1 \mapsto_{\omega_\alpha^e}^{r\_3} \mathcal{A}_2 \mapsto_{\omega_\alpha^e}^* \mathcal{A}_3 \mapsto_{\omega_\alpha^e}^{r\_4a} \mathcal{A}_4$ *then exists* $\mathcal{A}_2'$*,* $\mathcal{A}_3'$ *such that* $\mathcal{P}_0 \rhd \mathcal{A}_1 \mapsto_{\omega_\alpha^e}^* \mathcal{A}_2' \mapsto_{\omega_\alpha^e}^{r\_3} \mathcal{A}_3 \mapsto_{\omega_\alpha^e}^{r\_4a} \mathcal{A}_3' \mapsto_{\omega_\alpha^e}^* \mathcal{A}_4$
2. *For encoding rule instances* r_3 *and* r_4b*, we have* $\mathcal{P}_0 \rhd \mathcal{A}_1 \mapsto_{\omega_\alpha^e}^{r\_3} \mathcal{A}_2 \mapsto_{\omega_\alpha^e}^* \mathcal{A}_3 \mapsto_{\omega_\alpha^e}^{r\_4b} \mathcal{A}_4$ *then exists* $\mathcal{A}_2'$*,* $\mathcal{A}_3'$ *such that* $\mathcal{P}_0 \rhd \mathcal{A}_1 \mapsto_{\omega_\alpha^e}^* \mathcal{A}_2' \mapsto_{\omega_\alpha^e}^{r\_3} \mathcal{A}_3 \mapsto_{\omega_\alpha^e}^{r\_4b} \mathcal{A}_3' \mapsto_{\omega_\alpha^e}^* \mathcal{A}_4$

Theorem 6 asserts the soundness of the basic encoding: It states that ω_0^e derivations between commit free states of 0-neighbor restricted encodings have a mapping to ω_α^e derivations of its original 1-neighbor restricted program.

THEOREM 6 (Soundness of Basic Encoding). *Given a* 1*-neighbor restricted and locally quiescent CHR*^{e} *program* $\mathcal{P}_1$ *and a* 0*-neighbor restricted CHR*^{e} *program* $\mathcal{P}_0$ *such that* $\mathcal{P}_1 \rightsquigarrow_{1\mathrm{Nb}}^{\mathrm{basic}} \mathcal{P}_0$*, for any reachable states* Ω *and* Ω'*, if* $\mathcal{P}_0 \rhd \Omega \mapsto_{\omega_0^e}^* \Omega'$*, then we have either* Ω' *is not commit free or* $\mathcal{P}_1 \rhd \mathrm{DropSyncs}(\lceil\Omega\rceil; \mathcal{P}_1) \mapsto_{\omega_\alpha^e}^* \mathrm{DropSyncs}(\lceil\Omega''\rceil; \mathcal{P}_1)$.

7.2 Optimizations

We define optimized encoding schemes for two special cases. Consider the basic encoding scheme in Figure 13. Suppose that we apply to it a 1-neighbor restricted rule that is neighbor propagated (i.e., $S_y = \emptyset$) and furthermore P_y is persistent. We call such rules *neighbor persistent*. This encoding executes an unnecessary indirection of sending r_commit to neighbor Y in r_3 and completing the rule instance r with either r_4a or r_4b. However if $S_y = \emptyset$, primary location X can immediately complete the rule instance at r_3 without further communications with Y. This specialized encoding scheme is denoted by $\rightsquigarrow_{1\mathrm{Nb}}^{\mathrm{n-persist}}$ is shown in Figure 15. We also show an example rule *blend* which has this property ($S_y = \emptyset$ and we assume constraints $pallete(C)$ are persistent).

The next optimized encoding scheme applies specifically to 1-neighbor restricted rules which are not only primary propagated, but furthermore all propagated matching obligations are persistent[15]. We call such rules *primary persistent* rules. An example of this is the *trans* rule of the program of figure 1: Its primary matching obligation consists of only one propagated constraint $edge(Y, D)$ which is never deleted by any rule of the program. Figure 15 shows this specialized encoding as the function $\rightsquigarrow_{1\mathrm{Nb}}^{\mathrm{p-persist}}$. This optimized translation scheme is very similar to *rule localization* of link-restricted rule in distributed Datalog [8] and indeed it is a special case of rewriting where left-hand sides are not removed as a result of rule application.

Given a 1-neighbor restricted program $\mathcal{P}_1$, we define the translation function $\mathcal{P}_1 \rightsquigarrow_{1\mathrm{Nb}} \mathcal{P}_0$, where $\mathcal{P}_0$ is the 0-neighbor restricted program encoding in which we apply the optimized encodings where possible: We apply $\rightsquigarrow_{1\mathrm{Nb}}^{\mathrm{n-persist}}$ for 1-neighbor restricted rules in $\mathcal{P}_1$ that are neighbor persistent, while we apply $\rightsquigarrow_{1\mathrm{Nb}}^{\mathrm{p-persist}}$ for those that are primary persistent, and $\rightsquigarrow_{1\mathrm{Nb}}^{\mathrm{basic}}$ for all other 1-neighbor restricted rule.

We now consider the soundness of the optimized encoding (the soundness of the basic encoding has been shown in the previous section). In general, both the neighbor and primary persistent encoding schemes exploit a similar property: one location's matching obligation M of the 1-neighbor restricted rules is completely persistent and we do not need to lock and reserve any constraint in M. This simplifies the application of such 1-neighbor restricted rules and as long as we observe that persistent matching obligations M are fulfilled, the rule application can be completed entirely at the other location's discretion. Lemma 7 and 8 states this property for the neighbor persistent and primary persistent encoding schemes respectively.

LEMMA 7 (Neighbor Persistent Encoding Rule Serializability). *Given a* 1*-neighbor restricted and locally quiescent CHR*^{e} *program* $\mathcal{P}_1$ *and a* 0*-neighbor restricted CHR*^{e} *program* $\mathcal{P}_0$ *such that* $\mathcal{P}_1 \rightsquigarrow_{1\mathrm{Nb}} \mathcal{P}_0$ *and commit free abstract states* $\mathcal{A}_1$ $\mathcal{A}_2$*,* $\mathcal{A}_3$ *and* $\mathcal{A}_4$*, for some neighbor persistent encoding rule instances* r_2 *and* r_3*, we have* $\mathcal{P}_0 \rhd \mathcal{A}_1 \mapsto_{\omega_\alpha^e}^{r\_2} \mathcal{A}_2 \mapsto_{\omega_\alpha^e}^* \mathcal{A}_3 \mapsto_{\omega_\alpha^e}^{r\_3} \mathcal{A}_4$ *then there exists some* $\mathcal{A}_2'$ *such that* $\mathcal{P}_0 \rhd \mathcal{A}_1 \mapsto_{\omega_\alpha^e}^* \mathcal{A}_2' \mapsto_{\omega_\alpha^e}^{r\_2} \mathcal{A}_3 \mapsto_{\omega_\alpha^e}^{r\_3} \mathcal{A}_4$

[15] This corresponds to a rule in Figure 13 such that $P_x = \emptyset$ and $S_x = \emptyset$, hence we only have P_x' the persistent propagated rule heads.

$$trans : \boxed{[X]\, edge(Y, D)}, [Y]\, path(Z, D') \Longrightarrow X \neq Z \mid [X]\, path(Z, D + D')$$
$$\rightsquigarrow_{1\mathrm{Nb}}^{\mathrm{p-persist}} \left(\begin{array}{l} trans\_1 : [X]\, edge(Y, D) \Longrightarrow [Y]\, trans\_req(X, D) \\ trans\_2 : [Y]\, trans\_req(X, D), [Y]\, path(Z, D') \Longrightarrow X \neq Z \mid [X]\, path(Z, D + D') \end{array} \right)$$

$$(r : [X]\, P_x, [Y]\, P_y \setminus [Y]\, S_y \Longleftrightarrow G_x, G_y \mid B) \rightsquigarrow_{1\mathrm{Nb}}^{\mathrm{p-persist}} \left(\begin{array}{l} r\_1 : [X]\, P_x \Longrightarrow G_x \mid [Y]\, r\_req(Xs) \\ r\_2 : [Y]\, r\_req(Xs), [Y]\, P_y \setminus [Y]\, S_y \Longleftrightarrow G_y \mid B \end{array} \right)$$

where All $c \in P_x$ are persistent constraints
$Xs = \mathrm{FV}(P_x)$ and $\mathrm{FV}(G_x) \subseteq \mathrm{FV}(P_x)$ and $\mathrm{FV}(G_y) \subseteq \mathrm{FV}(P_x, P_y, S_y)$

Figure 15. Optimized Encodings for Primary Persistent Rules

LEMMA 8 (Primary Persistent Encoding Rule Serializability). *Given a 1-neighbor restricted and locally quiescent CHR^e program $\mathcal{P}_1$ and a 0-neighbor restricted CHR^e program $\mathcal{P}_0$ such that $\mathcal{P}_1 \rightsquigarrow_{1\mathrm{Nb}} \mathcal{P}_0$ and commit free abstract states $\mathcal{A}_1$ $\mathcal{A}_2$, $\mathcal{A}_3$ and $\mathcal{A}_4$, for some primary persistent encoding rule instances r_1 and r_2, we have $\mathcal{P}_0 \rhd \mathcal{A}_1 \mapsto_{\omega_\alpha^e}^{r\_1} \mathcal{A}_2 \mapsto_{\omega_\alpha^e}^{r\_2} \mathcal{A}_3 \mapsto_{\omega_\alpha^e}^{r\_2} \mathcal{A}_4$ then there exists some $\mathcal{A}_2'$ such that $\mathcal{P}_0 \rhd \mathcal{A}_1 \mapsto_{\omega_\alpha^e}^{*} \mathcal{A}_2' \mapsto_{\omega_\alpha^e}^{r\_1} \mathcal{A}_3 \mapsto_{\omega_\alpha^e}^{r\_2} \mathcal{A}_4$*

Theorem 9 states the soundness of the optimized encoding.

THEOREM 9 (Soundness of Optimized Encoding). *Given a 1-neighbor restricted and locally quiescent CHR^e program $\mathcal{P}_1$ and a 0-neighbor restricted CHR^e program $\mathcal{P}_0$ such that $\mathcal{P}_1 \rightsquigarrow_{1\mathrm{Nb}} \mathcal{P}_0$, for any reachable states Ω and Ω', if $\mathcal{P}_0 \rhd \Omega \mapsto_{\omega_0^e}^{*} \Omega'$, then we have either Ω' is not commit free or $\mathcal{P}_1 \rhd \mathrm{DropSyncs}(\lceil\Omega\rceil; \mathcal{P}_1) \mapsto_{\omega_\alpha^e}^{*} \mathrm{DropSyncs}(\lceil\Omega''\rceil; \mathcal{P}_1)$.*

8. Encoding n-Neighbor Restriction

We briefly discuss a generalized encoding scheme for n-neighbor restricted rules. The basic 1-neighbor restricted encoding of Figure 13 implements a consensus protocol between two nodes. Specifically, this encoding implements a *two-phase commit protocol* [14] led by an initial round of matching. Rules r_1 and r_2 represent the *matching phase*, while r_3 the *voting phase*, and r_4a and r_4b the *commit phase*. The encoding of n-neighbor restricted rules is then an implementation of a general consensus protocol that establishes consensus of a rule application among the primary location X (acting as the *coordinator* of the consensus) and n other directly connected and isolated neighbor locations Y_i for $i \in [1, n]$ (acting as the *cohorts* of the consensus). Details of the generalized encoding can be found in [6].

9. Conclusion and Future Works

We introduced CHR^e, an extension of CHR with located constraints and n-neighbor restricted rules for programming an ensemble of distributed computing entities. We defined the ω_α^e abstract semantics and ω_0^e operational semantics of CHR^e and showed their soundness. We gave an optimized encoding for 1-neighbor restricted rules into 0-neighbor restricted rules. Following this, we generalize this encoding scheme for n-neighbor restricted rules. We have developed a prototype implementation of CHR^e in Python with MPI (Message Passing Interface) as a proof of concept and demonstrated its relative scalability in distributed execution. In the future we intend to develop a more practical and competitive implementation of CHR^e in C or C++, imbued with existing CHR optimization techniques [13]. We also intend to explore an implementation over higher-level distributed graph processing frameworks like Google's Pregel [9]. Additionally, we intend to explore using ω_0^e to serve as an operational semantics that describes the core multiset rewriting fragment of Meld and derive extensions like aggregates and comprehensions as higher-level language encodings into ω_0^e. Our works on encoding n-neighbor restricted rules can be also applied to extend Meld.

References

[1] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein, P. Pillai, and J. D. Campbell. A Language for Large Ensembles of Independently Executing Nodes. In *Proc. of ICLP'09*, volume 5649, pages 265–280. Springer-Verlag, 2009.

[2] F. Cruz, M. P. Ashley-Rollman, S. C. Goldstein, Ricardo Rocha, and F. Pfenning. Bottom-Up Logic Programming for Multicores. In Vítor Santos Costa, editor, *Proc. of DAMP 2012*. ACM Digital Library, January 2012.

[3] G. J. Duck, P. J. Stuckey, M. Garcia de la Banda, and C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *In 20th Int. Conf. on Logic Programming ICLP'04*, pages 90–104. Springer, 2004.

[4] T. Frühwirth. Parallelizing Union-find in Constraint Handling Rules using Confluence Analysis. In *Logic Programming: 21st Int. Conf. ICLP 2005, Vol. 3668 of Lecture Notes in Computer Science*, pages 113–127. Springer, 2005.

[5] Thom Frhwirth. *Constraint Handling Rules*. Cambridge University Press, ISBN 9780521877763, 2009.

[6] E. S. L. Lam and I. Cervesato. Decentralized Execution of Constraint Handling Rules for Ensembles (Extended-Version). Technical Report CMU-CS-13-106/CMU-CS-QTR-118, Carnegie Mellon University, Apr 2013.

[7] E. S. L. Lam and M. Sulzmann. Concurrent Goal-based Execution of Constraint Handling Rules. *TPLP*, 11(6):841–879, 2011.

[8] 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 *Proc. of SIGMOD '06*, pages 97–108. ACM, 2006.

[9] G. Malewicz, M. H. Austern, A. J.C Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proc. of Int. Conf. on Management of data*, SIGMOD, pages 135–146, USA, 2010. ACM.

[10] V. Nigam, L. Jia, B. T. Loo, and A. Scedrov. Maintaining distributed logic programs incrementally. In *Proc. of PPDP'11*, pages 125–136. ACM, 2011.

[11] F. Pfenning. Substructural Operational Semantics and Linear Destination-Passing Style (Invited Talk). In *APLAS*, page 196, 2004.

[12] M. J. Quinn. Analysis and Benchmarking of Two Parallel Sorting Algorithms: Hyperquicksort and Quickmerge. *BIT*, 29(2):239–250, June 1989.

[13] T. Schrijvers. Analyses, Optimizations and Extensions of Constraint Handling Rules: Ph.D. Summary. In *ICLP*, pages 435–436, 2005.

[14] D. Skeen and M. Stonebraker. A Formal Model of Crash Recovery in a Distributed Systems. *IEEE Transactions on Software Engineering*, pages 219–228, 1983.

[15] A. Triossi, S. Orlando, A. Raffaetà, and T. Frühwirth. Compiling CHR to parallel hardware. In *Proc. of PPDP'12*, pages 173–184. ACM, 2012.