Exhaustive Parallel Rewriting with Multiple Removals

Frank Raiser and Thom Frühwirth

Faculty of Engineering and Computer Science, Ulm University, Germany
firstname.lastname@uni-ulm.de

Abstract. Parallel multiset rewriting is usually restricted to be free of overlaps, such that multiple rule applications cannot remove the same object. In this work, we present a parallel execution strategy for Constraint Handling Rules that allows multiple removals of constraints, for which different multiplicities have no effect on results. We show that the resulting operational semantics is sound with respect to sequential execution and discuss exemplary applications.

1 Introduction

Rewriting is an important subject in computer science and numerous flavors of rewriting systems (term rewriting, graph rewriting, etc.) have been developed. Their non-deterministic application of rules facilitates parallel execution strategies. Modern processors and graphics cards, with their large number of cores, further increase demand for parallel rule application. However, multiple rules applied in parallel can lead to two kinds of conflicts: Firstly, rule applications may need to remove the same object (i.e., term, graph node, etc.), and secondly, an object removed by one rule application may be required by another.

This work focuses on multiset rewriting of objects, as found in Gamma [1], the chemical abstract machine (CHAM) [2], or Constraint Handling Rules (CHR) [3]. We investigate parallelism with respect to special kinds of objects: for *multiplicity-independent* objects their multiplicity has no effect on the computed answer. This fact can be exploited and leads to a new parallel rewriting approach without the above conflicts.

In the literature, however, many implementations restrict parallelism such that these conflicts do not occur. For example, in [4] rewriting is performed on independent parts of a graph. More recent work, like [5], approaches massively parallel graph rewriting, but still performs independent rewriting steps. In [5] multiple transformation processes of the same graph are executed in parallel such that each process may select different non-deterministic possibilities.

We investigate our approach for Constraint Handling Rules (CHR) [3], which is a committed-choice multiset rewriting language. It performs exhaustive, forward chaining, multiset constraint rewriting and furthermore offers a first-order logical reading, as well as a linear logical reading [3]. Additionally, it provides support for built-in constraints and a corresponding built-in constraint theory.

The potential for parallelism inherent in CHR has already been recognized during its early development. Recent results include parallel union-find [6] and preflow-push [7] implementations, which have been formulated on an abstract level, again assuming non-overlapping rewriting.

A pragmatic approach to rewriting of overlapping objects in CHR has been presented in [8]: based on Haskell's support for shared transaction memory, rule applications are evaluated in parallel and committed to the store. In case of two rules that want to remove the same constraint, only the first rule's commit gets executed. The second commit fails and is rolled back, i.e. the rule application is hindered. A related approach is given in [9], in which an atomic operational semantics for CHR is given.

The core idea of our work is as follows: The semantics of multiplicity-independent objects allows us to consider additional copies of them, such that a duplicate removal of one can be replaced by removing two distinct objects. Therefore, our proposed parallel execution strategy is free of these removal conflicts, while still remaining sound. As additional distinct objects are not explicitly created, our approach can also be interpreted as allowing multiple removals of an object within one parallel step.

Example 1. Consider the following CHR program for computing prime numbers.

$$\text{prime}(N) \setminus \text{prime}(M) \Leftrightarrow M\%N = 0 \mid \top$$

Without going into details of an execution strategy, the core idea of this program is as follows: candidates for prime numbers are encoded as prime(i) constraints for some number i. A candidate with number i can then remove all candidates with numbers that are multiples of i, similar to the sieve of Eratosthenes. Having a number i available as a prime candidate multiple times makes no difference: if it is not prime, then the candidate with a prime factor of i will remove all of its occurrences, otherwise additional copies redundantly denote that i is a prime number.

To demonstrate the effect of our proposed parallel execution strategy, consider the non-prime number 30 encoded in prime(30), that has multiple prime factors. Hence, the above rule can remove prime(30) via prime(2), prime(3), prime(4), prime(5), prime(6), prime(10), and prime(15). With only one prime(30) constraint available, this leads to several possible rule applications that require the removal of the same object. With our approach instead, we remove the prime(30) constraint multiple times – once, for each of the possible rule applications.

A consequence of this approach is that all applicable rules can be applied in one parallel step. We show that this execution strategy is sound with respect to sequential execution. As this work is a first foray into massive parallelism for CHR, we assume an unlimited number of processors to be available, nevertheless we show that execution with any finite number of processors is still sound. We present CHR programs for the computation of minima, maxima, prime numbers, and sorting, that require only a constant number of parallel rule applications.

These results, based on an unlimited number of processors, are consistent with similar ones in the literature. For example, [10] presents constant-time parallel sorting and [5] discusses how NP-complete problems can be solved in polynomial time with a massive number of processors.

Clearly not all objects in multiset rewriting have this multiplicity-independent nature. However, the presented programs are essential algorithms found in many applications. It may thus be desirable to create a parallel execution strategy that allows for both, our proposed rewriting strategy and another strategy from existing literature.

The remainder of this work is structured as follows. After introducing the operational semantics of CHR in Section 2, we present our set-based parallel formulation in Section 3. In that section we further present applications and discuss properties of the proposed parallel execution scheme, in particular its soundness with respect to sequential execution. Finally, we conclude in Section 4 and give an overview of possible further research directions.

2 Preliminaries

In this section, we present the equivalence-based operational semantics ω_e for Constraint Handling Rules [11]. It corresponds to the *very abstract operational semantics* ω_{va} in [3]. We chose this formulation, because it clearly separates *CHR constraints* (also called *user-defined constraints*) which are rewritten, from *built-in constraints*, which are handled by a constraint theory $\mathcal{CT}$.

Formally, CHR is a state transition system, hence we begin with the definition of a state.

Definition 1 (State). *A* (CHR) state σ *is a tuple* $\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle$. *The* user-defined (constraint) store $\mathbb{G}$ *is a multiset of CHR constraints. The* built-in (constraint) store $\mathbb{B}$ *is a conjunction of built-in constraints.* $\mathbb{V}$ *is a set of variables, called the* global variables. *We use* Σ_e *to denote the set of all states.*

For clarity we usually omit curly brackets and denote (multi-)set union with comma, i.e. $\langle \{a\} \uplus \{b\}; \top; \emptyset \rangle = \langle a, b; \top; \emptyset \rangle$. A CHR state contains different kinds of variables, defined as follows.

Definition 2 (Variable Types). *For the variables occurring in a state* $\sigma = \langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle$ *we distinguish three different types:*

1. *a variable* $v \in \mathbb{V}$ *is called a* global *variable*
2. *a variable* $v \notin \mathbb{V}$ *is called a* local *variable*
3. *a variable* $v \notin (\mathbb{V} \cup \mathbb{G})$ *is called a* strictly local *variable*

The equivalence-based operational semantics is founded on the following equivalence relation between CHR states. State equivalence and the resulting transition system are discussed in more detail in [11].

Definition 3 (State Equivalence).
Equivalence between CHR states is the smallest equivalence relation $\equiv$ over CHR states that satisfies the following conditions:

1. (Equality as Substitution) $\langle \mathbb{G}; x \doteq t \wedge \mathbb{B}; \mathbb{V} \rangle \equiv \langle \mathbb{G}\,[x/t]\,; x \doteq t \wedge \mathbb{B}; \mathbb{V} \rangle$
2. (Transformation of the Constraint Store) *If* $\mathcal{CT} \models \exists \bar{s}.\mathbb{B} \leftrightarrow \exists \bar{s}'.\mathbb{B}'$ *where* $\bar{s}, \bar{s}'$ *are the strictly local variables of* $\mathbb{B}, \mathbb{B}'$*, respectively, then:* $\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle \equiv \langle \mathbb{G}; \mathbb{B}'; \mathbb{V} \rangle$
3. (Omission of Non-Occurring Global Variables) *If* x *is a variable that does not occur in* $\mathbb{G}$ *or* $\mathbb{B}$ *then:* $\langle \mathbb{G}; \mathbb{B}; \{x\} \cup \mathbb{V} \rangle \equiv \langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle$
4. (Equivalence of Failed States) $\langle \mathbb{G}; \bot; \mathbb{V} \rangle \equiv \langle \mathbb{G}'; \bot; \mathbb{V} \rangle$

The following formulation of transitions regards CHR as a rewriting system for equivalence classes of states. Note that we freely mix equivalence classes and their representatives, though, i.e. we often write $\sigma \rightarrowtail \tau$ instead of $[\sigma] \rightarrowtail [\tau]$. Syntactically, a CHR rule is of the form $r \ @\ H_1 \backslash H_2 \Leftrightarrow G \mid B_c, B_b$ where r is an optional *rulename*, H_1 is the *kept head*, H_2 the *removed head*, G the *guard*, and the *body* consists of user-defined constraints B_c and built-in constraints B_b.

A *variant* of a rule $(r \ @\ H_1 \backslash H_2 \Leftrightarrow G \mid B_c, B_b)$ with variables $\bar{x}$ is a rule of the form $(r \ @\ H_1 \backslash H_2 \Leftrightarrow G \mid B_c, B_b)[\bar{x}/\bar{y}]$ for any sequence of pairwise distinct variables $\bar{y}$. For any rule $(r \ @\ H_1 \backslash H_2 \Leftrightarrow G \mid B_c, B_b)$, its *local variables* $\bar{l}_r$ are defined as $\bar{l}_r ::= \mathrm{vars}(G, B_c, B_b) \setminus \mathrm{vars}(H_1, H_2)$.

Definition 4 (Transitions).
For a CHR program $\mathcal{P}$*, the state transition system* $(\Sigma_e/\equiv, \rightarrowtail)$ *is defined as follows. The transition is based on a variant of a rule* r *in* $\mathcal{P}$ *such that its local variables are disjoint from the variables occurring in the pre-transition state.*

$$\frac{r \ @\ H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b}{[\langle H_1, H_2, \mathbb{G}; G \wedge \mathbb{B}; \mathbb{V} \rangle] \rightarrowtail^r [\langle H_1, B_c, \mathbb{G}; G \wedge B_b \wedge \mathbb{B}; \mathbb{V} \rangle]}$$

3 Exhaustive Parallel Execution

In this section, we present our proposal for a parallel execution strategy suitable for multiplicity-independent objects. After formally introducing our proposed parallel operational semantics CHR^{mp}, Section 3.1 presents possible applications. Section 3.2 then shows its soundness with respect to sequential execution.

Based on the definition of CHR states, we first define a CHR^{mp} state that adheres to set-semantics.

Definition 5 (CHR^{mp} state). *A* CHR^{mp} state *is a tuple* $\langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle$ *with a set* $\mathbb{G}$ *of CHR constraints, a conjunction* $\mathbb{B}$ *of built-in constraints, and a set of global variables* $\mathbb{V}$*. The set* $\mathbb{G}$ *is considered modulo* $\mathbb{B}$*, i.e. for different constraints* $c(\bar{x_1}), c(\bar{x_2}) \in \mathbb{G}$ *holds that* $\mathcal{CT} \not\models \mathbb{B} \rightarrow (c(\bar{x_1}) = c(\bar{x_2}))$*.*

The equivalence relation $\equiv$ *between CHR states directly transfers to CHR^{mp} states, with the exception that failed states may not turn the set of CHR constraints into a multiset under* $\equiv$*.*

Finally, the transition relation $\rightarrowtail$ applies to CHR^{mp} states as defined for CHR states.[1]

Example 2. The CHR state $\langle c(X), c(3); X = 3; \emptyset\rangle$ is not a CHR^{mp} state, as $X = 3 \rightarrow c(X) = c(3)$. We can turn any CHR state into a CHR^{mp} state by eliminating duplicate CHR constraints, hence creating a set from the multiset of CHR constraints: $\langle c(X); X = 3; \emptyset\rangle \equiv \langle c(3); \top; \emptyset\rangle$.

We now define the parallel state transition system of CHR^{mp}. It works as follows: for all considered sequential rule applications the removed constraints are removed from the state and all bodies are added. As the constraints are kept in a set we have no means to refer to a specific constraint. Therefore, we build upon state equivalence $\equiv$ and consider a conjunction of all involved built-in stores, assuming rule variants with distinct variables.

Definition 6. *Let $\sigma = \langle \mathbb{G}; \mathbb{B}; \mathbb{V}\rangle$ be a CHR^{mp} state and let $\mathcal{R}$ be the smallest set such that for each rule r @ $H_1 \backslash H_2 \Leftrightarrow G \mid B_c, B_b$ with $\sigma \equiv \langle H_1, H_2, \mathbb{G}'; G \wedge \mathbb{B}'; \mathbb{V}\rangle$ holds $(H_1, H_2, B_c, B_b, \mathbb{B}') \in \mathcal{R}$. We then define for $R \subseteq \mathcal{R}$:*

- *the set of removed constraints: $D = \{c \mid \exists(\_, H_2, \_, \_, \mathbb{B}') \in R, c \in \mathbb{G} : \mathcal{CT} \models (H_2 \wedge \mathbb{B}') \rightarrow c\}$*
- *the set of added constraints: $A = \{c \mid \exists(\_, \_, B_c, \_, \_) \in R : c \in B_c\}$*
- *the conjunction of added built-in constraints: $B = \bigwedge_{(\_,\_,\_,B_b,\mathbb{B}')\in R} \mathbb{B}' \wedge B_b$*

A parallel transition (step) *of σ is defined as:*

$$\sigma \twoheadrightarrow^R \langle (\mathbb{G} \setminus D) \cup A; \mathbb{B} \wedge B; \mathbb{V}\rangle$$

If the specific set R is not of importance we also write $\twoheadrightarrow$ instead of $\twoheadrightarrow^R$.

Example 3. Reconsider the CHR program for computing prime numbers and the state $\sigma = \langle \text{prime}(2), \text{prime}(3), \text{prime}(4), \text{prime}(5), \text{prime}(X); X = 6; \{X\}\rangle$.

There are a total of three possible rule applications, removing the non-prime numbers 4 and 6, where 6 is removed twice. We therefore have the following:

$$\begin{aligned}
\sigma &\equiv \langle \text{prime}(N_1), \text{prime}(M_1), \ldots; X = 6 \wedge N_1 = 2 \wedge M_1 = 4; \{X\}\rangle \\
\sigma &\equiv \langle \text{prime}(N_2), \text{prime}(M_2), \ldots; X = 6 \wedge N_2 = 2 \wedge M_2 = 6; \{X\}\rangle \\
\sigma &\equiv \langle \text{prime}(N_3), \text{prime}(M_3), \ldots; X = 6 \wedge N_3 = 3 \wedge M_3 = 6; \{X\}\rangle
\end{aligned}$$

$$\Rightarrow \mathcal{R} = \left\{ \begin{array}{l} (\{\text{prime}(N_1)\}, \{\text{prime}(M_1)\}, \emptyset, \top, X = 6 \wedge N_1 = 2 \wedge M_1 = 4), \\ (\{\text{prime}(N_2)\}, \{\text{prime}(M_2)\}, \emptyset, \top, X = 6 \wedge N_2 = 2 \wedge M_2 = 6), \\ (\{\text{prime}(N_3)\}, \{\text{prime}(M_3)\}, \emptyset, \top, X = 6 \wedge N_3 = 3 \wedge M_3 = 6) \end{array} \right\}$$

We can now perform all three possible rule applications in parallel, i.e. $R = \mathcal{R}$, resulting in the following sets: $D = \{\text{prime}(4), \text{prime}(X)\}$, $A = \emptyset$, and $B = (X =$

[1] Rule heads may require a multiset for matching, in which case additional rules can be considered, which overlap these multiset constraints to make them matchable by a single constraint.

$6 \wedge N_1 = 2 \wedge M_1 = 4) \wedge (X = 6 \wedge N_2 = 2 \wedge M_2 = 6) \wedge (X = 6 \wedge N_3 = 3 \wedge M_3 = 6)$. This leads to the parallel step:

$$\begin{aligned} \sigma &\twoheadrightarrow^{\mathcal{R}} \langle \text{prime}(2), \text{prime}(3), \text{prime}(5); X = 6 \wedge B; \{X\} \rangle \\ &\equiv \quad \langle \text{prime}(2), \text{prime}(3), \text{prime}(5); X = 6; \{X\} \rangle \end{aligned}$$

Hence, a single parallel step is sufficient to filter all non-prime numbers. The following section presents further examples of such filter algorithms.

3.1 Applications

In this section, we examine different applications and the effect of executing these programs in CHR^{mp}. All constraints in this section have multiplicity-independent semantics, such that adding additional copies does not affect results.

Filter Programs There exists a class of programs that filter constraints from a set of available constraints based on some given criteria. Typical representatives of this class are programs for computing minima, maxima, or prime numbers. They usually consist of a single rule, that takes two constraints and removes one of them if the guard is satisfied. For example, the following two rules compute the minimum and maximum of a set of available numbers, respectively.

$$\begin{aligned} &\min(A) \backslash \min(B) \Leftrightarrow A < B \mid \top \\ &\max(A) \backslash \max(B) \Leftrightarrow A > B \mid \top \end{aligned}$$

Executing these programs with CHR^{mp} requires only a single exhaustive parallel step: Assume that there are n min /1-constraints in the store, then there exists one such constraint $\min(n_0)$ such that n_0 is the smallest of the n arguments. In the original state this constraint can be used to remove any other min /1-constraint. Therefore, in CHR^{mp} all of these removals are combined into one exhaustive parallel step, leaving only $\min(n_0)$ as a result.

The computations made for such filter programs are sound with respect to sequential execution if the programs are deletion-acyclic, according to Definition 7. For the above two programs this is straightforward, because e.g., min /1-constraints can only be removed by ones with a smaller argument.

Sorting A more interesting application of CHR^{mp} is to perform exhaustive parallel sorting. The following rule performs sorting in CHR. It assumes the distinct input numbers $n_1, \ldots, n_N$ as constraints of the form $0 \rightsquigarrow n_i \forall 1 \leq i \leq N$:

$$A \rightsquigarrow B \backslash A \rightsquigarrow C \Leftrightarrow B < C \mid B \rightsquigarrow C$$

The following is a sequential computation to sort the numbers 1,. . . ,4:

$$\begin{aligned}
&\langle \underline{0 \rightsquigarrow 1}, \underline{0 \rightsquigarrow 3}, 0 \rightsquigarrow 2, 0 \rightsquigarrow 4; \top; \emptyset \rangle \\
\rightarrowtail\ &\langle \underline{0 \rightsquigarrow 1}, 1 \rightsquigarrow 3, \underline{0 \rightsquigarrow 2}, 0 \rightsquigarrow 4; \top; \emptyset \rangle \\
\rightarrowtail\ &\langle 0 \rightsquigarrow 1, \underline{1 \rightsquigarrow 3}, \underline{1 \rightsquigarrow 2}, 0 \rightsquigarrow 4; \top; \emptyset \rangle \\
\rightarrowtail\ &\langle \underline{0 \rightsquigarrow 1}, 1 \rightsquigarrow 2, 2 \rightsquigarrow 3, \underline{0 \rightsquigarrow 4}; \top; \emptyset \rangle \\
\rightarrowtail\ &\langle 0 \rightsquigarrow 1, \underline{1 \rightsquigarrow 2}, 2 \rightsquigarrow 3, \underline{1 \rightsquigarrow 4}; \top; \emptyset \rangle \\
\rightarrowtail\ &\langle 0 \rightsquigarrow 1, 1 \rightsquigarrow 2, \underline{2 \rightsquigarrow 3}, \underline{2 \rightsquigarrow 4}; \top; \emptyset \rangle \\
\rightarrowtail\ &\langle 0 \rightsquigarrow 1, 1 \rightsquigarrow 2, 2 \rightsquigarrow 3, 3 \rightsquigarrow 4; \top; \emptyset \rangle
\end{aligned}$$

Executing this program without any changes in CHR^{mp} results in the following computation:

$$\begin{aligned}
&\langle 0 \rightsquigarrow 1, 0 \rightsquigarrow 3, 0 \rightsquigarrow 2, 0 \rightsquigarrow 4; \top; \emptyset \rangle \\
\twoheadrightarrow\ &\langle 0 \rightsquigarrow 1, 1 \rightsquigarrow 3, 1 \rightsquigarrow 2, 1 \rightsquigarrow 4, 2 \rightsquigarrow 3, 2 \rightsquigarrow 4, 3 \rightsquigarrow 4; \top; \emptyset \rangle \\
\twoheadrightarrow\ &\langle 0 \rightsquigarrow 1, 1 \rightsquigarrow 2, 2 \rightsquigarrow 3, 2 \rightsquigarrow 4, 3 \rightsquigarrow 4; \top; \emptyset \rangle \\
\twoheadrightarrow\ &\langle 0 \rightsquigarrow 1, 1 \rightsquigarrow 2, 2 \rightsquigarrow 3, 3 \rightsquigarrow 4; \top; \emptyset \rangle
\end{aligned}$$

Here, we add constraints, e.g., $2 \rightsquigarrow 4$, that have already been removed in the same step, resulting in an unwanted reintroduction. Hence, the first parallel step generates $\mathcal{O}(N^2)$ constraints of the form $n_i \rightsquigarrow n_j$ with $i < j$. There remains exactly one constraint $0 \rightsquigarrow n_k$ such that n_k is the smallest number. In the second step, the second smallest number is computed and again $\mathcal{O}(N^2)$ constraints are regenerated. Therefore, all computations involving larger numbers than the currently minimal one are wasted and a total of $\mathcal{O}(N)$ parallel steps are required.

This can be improved upon by slightly modifying the program such that it becomes a filter program. The idea is to compare all numbers with all other numbers in one step, and then compute the minimum of all comparisons for each number. This is realized by the following program:

$$\begin{aligned}
&A \rightsquigarrow B, A \rightsquigarrow C \Leftrightarrow B < C \mid A \triangleleft B, B \triangleleft C \\
&A \triangleleft B \backslash A \triangleleft C \Leftrightarrow B < C \mid \top
\end{aligned}$$

The computation is split into two phases, represented by the constraint symbols $\rightsquigarrow$ and $\triangleleft$. Below is the execution of the program in CHR^{mp}, again for the numbers $1, \ldots, 4$.

$$\begin{aligned}
&\langle 0 \rightsquigarrow 1, 0 \rightsquigarrow 3, 0 \rightsquigarrow 2, 0 \rightsquigarrow 4; \top; \emptyset \rangle \\
\twoheadrightarrow\ &\langle 0 \triangleleft 1, 0 \triangleleft 2, 0 \triangleleft 3, 1 \triangleleft 2, 1 \triangleleft 3, 1 \triangleleft 4, 2 \triangleleft 3, 2 \triangleleft 4, 3 \triangleleft 4; \top; \emptyset \rangle \\
\twoheadrightarrow\ &\langle 0 \triangleleft 1, 1 \triangleleft 2, 2 \triangleleft 3, 3 \triangleleft 4; \top; \emptyset \rangle
\end{aligned}$$

After this modification the program is able to compute all $\triangleleft$-pairs in one parallel step and filter unwanted pairs in a second step. Therefore, sorting any amount of numbers requires a constant number of transitions. As the first rule generates $\mathcal{O}(N^2)$ $\triangleleft$-pairs, $\mathcal{O}(N^4)$ processors are required for filtering. Again, we find the typical trade-off that requires more processors in order to reduce runtime complexity.

3.2 Soundness of CHR^{mp}

While the CHR^{mp} execution strategy is worthwhile on its own, it may sometimes be desirable to ensure that a corresponding computation exists on a sequential processor. Hence, in this section we investigate soundness of CHR^{mp} with respect to sequential execution. In general, this is not always possible as the following example shows.

Example 4. Consider the program with the rule $c(X) \backslash c(Y) \Leftrightarrow \top$. An application of the rule removes a c-constraint only when another one is present in the store. Hence, the final result of exhaustively applying this rule sequentially contains a single remaining c-constraint (or none, if the initial goal contains no c-constraints).

Contrary to that, consider the state $\sigma = \langle c(1), c(2); \top; \emptyset \rangle$, which allows for the following CHR^{mp} computation: $\sigma \twoheadrightarrow^{\mathcal{R}} \langle \emptyset; \top; \emptyset \rangle$. Thus, as both c-constraints could potentially be removed, the exhaustive parallel execution strategy removes all of them – resulting in a state that is not reachable by sequential computation.

The above example shows that a program, in which two constraints are responsible for their mutual removal, is no longer sound in CHR^{mp} with respect to sequential execution. However, we can get soundness for the subset of programs which allow no mutual removal. More precisely, soundness requires that the programs are deletion-acyclic according to the following definition. The binary relation $\mathcal{D}(\sigma, R)$ represents a deletion dependency, i.e. $(c, d) \in \mathcal{D}(\sigma, R)$ means the constraint c is required to remove constraint d when applying one of the rules in R.

Definition 7 (Deletion Dependency, Deletion-Acyclic).
Let $\sigma = \langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle \twoheadrightarrow^{R} \tau$, then the deletion dependency $\mathcal{D}(\sigma, R)$ *is a binary relation such that $(c, d) \in \mathcal{D}(\sigma, R)$ if and only if $c, d \in \mathbb{G}$ and there exist $(H_1, H_2, B_c, B_b, \mathbb{B}') \in R$ and $c' \in H_1, d' \in H_2$ such that $c' \wedge \mathbb{B}' \rightarrow c$ and $d' \wedge \mathbb{B}' \rightarrow d$ or in the case of $H_1 = \emptyset$ there exists $d' \in H_2$ such that $d' \wedge \mathbb{B}' \rightarrow d$ and $(\cdot, d) \in \mathcal{D}(\sigma, R)$, where $\cdot$ is a unique symbol not occurring in the program.*

A CHR^{mp} program $\mathcal{P}$ is deletion-acyclic *if and only if for all σ such that $\sigma \twoheadrightarrow^{\mathcal{R}} \tau$ the transitive closure $\mathcal{D}(\sigma, \mathcal{R})^+$ is irreflexive.*

The following soundness result requires a deletion-acyclic program and to account for the set-semantics of CHR^{mp} the program for sequential execution is extended by well-known set-semantics rules of the kind $c(\bar{x}) \backslash c(\bar{x}) \Leftrightarrow \top$. This theorem shows soundness for $R \subseteq \mathcal{R}$, hence, it also holds for a non-exhaustive number of parallel rule applications, as would be the case with a finite number of processors.

Theorem 1 (Soundness). *Let $\mathcal{P}$ be a deletion-acyclic CHR^{mp} program and $\mathcal{P}'$ be the CHR program $\mathcal{P}$ extended with set-semantics rule. Let $\sigma = \langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle \twoheadrightarrow^{R}_{\mathcal{P}} \tau$ with $\mathbb{V} = \mathrm{vars}(\mathbb{G}, \mathbb{B})$, then there exists a multiset $\mathbb{G}'$ with $c \in \mathbb{G}' \Rightarrow c \in \mathbb{G}$ such that $\sigma' = \langle \mathbb{G}'; \mathbb{B}; \mathbb{V} \rangle \rightarrowtail^{|R|}_{\mathcal{P}'} \tau' \rightarrowtail^{*} \tau$, where the first $|R|$ rule applications coincide with those in R and the latter rule applications only use set-semantics rules.*

Example 5. Reconsider the above example computation for computing prime numbers. The program is deletion-acyclic as every number only removes multiples of itself. For the above state σ we see that

$$\mathcal{D}(\sigma, \mathcal{R}) = \{(\text{prime}(2), \text{prime}(4)), (\text{prime}(2), \text{prime}(X)), (\text{prime}(3), \text{prime}(X))\}$$

Hence, the need to duplicate $\text{prime}(X)$ for a sequential execution. The following is the sequential execution according to Theorem 1, for $\mathbb{G} = \text{prime}(4), \text{prime}(5)$.

$$\begin{aligned}
\sigma' = \ & \langle \underline{\text{prime}(2)}, \text{prime}(3), \mathbb{G}, \underline{\text{prime}(X)}, \text{prime}(X); X = 6; \{X\} \rangle \\
\rightarrowtail_{\mathcal{P}'} \ & \langle \underline{\text{prime}(2)}, \underline{\text{prime}(3)}, \mathbb{G}, \underline{\text{prime}(X)}; X = 6; \{X\} \rangle \\
\rightarrowtail_{\mathcal{P}'} \ & \langle \underline{\text{prime}(2)}, \underline{\text{prime}(3)}, \underline{\text{prime}(4)}, \text{prime}(5); X = 6; \{X\} \rangle \\
\rightarrowtail_{\mathcal{P}'} \ & \langle \underline{\text{prime}(2)}, \text{prime}(3), \underline{\text{prime}(5)}; X = 6; \{X\} \rangle
\end{aligned}$$

Note that extending the initial state to also contain prime(7) and prime(8) implies that a correct order has to be chosen for the sequential execution. As both, prime(2) and prime(4), are able to remove prime(8) there are two copies of prime(8). However, as prime(2) can also remove prime(4) it has to be ensured that this is done only after prime(4) has removed one prime(8) constraint. The induction used in the proof of Theorem 1 guarantees the existence of such an order for deletion-acyclic programs.

4 Conclusion and Future Work

In this work, we presented CHR^{mp}, a parallel execution strategy for Constraint Handling Rules. It is based on the notion of multiplicity-independent objects, for which different multiplicities do not affect results. We have shown how this can be exploited to allow multiple removal of objects by parallel rule applications.

We have proven that the resulting operational semantics is sound with respect to sequential execution for deletion-acyclic programs. In formalizing CHR^{mp} we referred to an unlimited number of processors, yet we have also shown that an execution strategy based on a finite number is equally sound. Finally, we have given example programs for filtering and sorting in order to demonstrate the viability of CHR^{mp}.

In this work, we concentrated on the formal aspects of CHR^{mp}, thus abstracting from more specific details. Multiple removal of constraints by different rule applications was shown to be possible and worthwhile, however, we have not yet investigated implementation details for our approach. We assume that for optimal efficiency a concurrent-read concurrent-write (CRCW) RAM architecture will be required.

Similarly, we abstracted from treatment of built-in constraints. Built-in constraints from all parallel rule applications must be combined efficiently, again implying the need for a CRCW RAM architecture and an optimized union-find implementation.

CHR^{mp} relies on multiplicity-independent objects, yet most programs require other forms of objects as well. Therefore, we plan to investigate the possibility

of considering CHR^{mp} in a modular fashion, such that it can be combined with other existing operational semantics. Hence, a program could work with a traditional CHR implementation, yet refer to CHR^{mp} as a module for sorting a sequence of objects. A similar approach would be the possibility to explicitly mark constraints as multiplicity-independent in the source code, thus making them subject to a CHR^{mp}-based execution strategy.

Finally, the current formulation of CHR^{mp} is non-terminating for propagation rules. We deliberately left this aspect undefined, as there are different approaches available to alleviate this problem. One of these is the introduction of *persistent constraints*, recently presented in [12], which are multiplicity-independent objects by their nature. Therefore, the combination of CHR^{mp} and persistent constraints appears promising.

References

1. Banâtre, J.P., Le Métayer, D.: Programming by multiset transformation. Communications of the ACM **36**(1) (1993) 98–111
2. Berry, G., Boudol, G.: The chemical abstract machine. In: POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM (1990) 81–94
3. Frühwirth, T.: Constraint Handling Rules. Cambridge University Press (2009)
4. Bauderon, M.: Parallel rewriting of graphs through the pullback approach. Electronic Notes in Theoretical Computer Science **2** (1995)
5. Kreowski, H.J., Kuske, S.: Graph multiset transformation as a framework for massively parallel computation. In Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G., eds.: Graph Transformations, 4th International Conference, ICGT. Volume 5214 of Lecture Notes in Computer Science., Springer-Verlag (2008) 351–365
6. Frühwirth, T.: Parallelizing union-find in constraint handling rules using confluence analysis. In: Principles and Practice of Constraint Programming, 11th International Conference, CP 2005, Sitges, Spain (October 2005)
7. Meister, M.: Concurrency of the preflow-push algorithm in Constraint Handling Rules. In Fages, F., Rossi, F., Soliman, S., eds.: Constraint Solving and Constraint Logic Programming, Annual ERCIM Workshop, CSCLP 2007, Rocquencourt, France (2007) 160–169
8. Sulzmann, M., Lam, E.S.L.: Parallel execution of multi-set constraint rewrite rules. In Antoy, S., Albert, E., eds.: Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP), Valencia, Spain, ACM (July 2008) 20–31
9. Schrijvers, T., Sulzmann, M.: Transactions in constraint handling rules. In: Logic Programming, 24th International Conference, ICLP 2008, Berlin, Heidelberg, Springer-Verlag (2008) 516–530
10. Gasarch, W., Golub, E., Kruskal, C.: Constant time parallel sorting: an empirical view. Journal of Computer and System Sciences **67**(1) (2003) 63–91
11. Raiser, F., Betz, H., Frühwirth, T.: Equivalence of CHR states revisited. In Raiser, F., Sneyers, J., eds.: 6th International Workshop on Constraint Handling Rules (CHR). (2009) 34–48
12. Betz, H., Raiser, F., Frühwirth, T.: A complete and terminating execution model for Constraint Handling Rules. Theory and Practice of Logic Programming **10**(4-6) (2010) 597–610

A Proofs

Theorem 2 (Soundness). *Let $\mathcal{P}$ be a deletion-acyclic CHR^{mp} program and $\mathcal{P}'$ be the CHR program $\mathcal{P}$ extended with set-semantics rule. Let $\sigma = \langle \mathbb{G}; \mathbb{B}; \mathbb{V} \rangle \twoheadrightarrow^{R}_{\mathcal{P}} \tau$ with $\mathbb{V} = \mathrm{vars}(\mathbb{G}, \mathbb{B})$, then there exists a multiset $\mathbb{G}'$ with $c \in \mathbb{G}' \Rightarrow c \in \mathbb{G}$ such that $\sigma' = \langle \mathbb{G}'; \mathbb{B}; \mathbb{V} \rangle \rightarrowtail^{|R|}_{\mathcal{P}'} \tau' \rightarrowtail^{*} \tau$, where the first $|R|$ rule applications coincide with those in R and the latter rule applications only use set-semantics rules.*

Proof. In the following proof let $\#(c, \mathbb{G})$ denote the multiplicity of the constraint c in the multiset $\mathbb{G}$. For the remainder of the proof we assume w.l.o.g. that the built-in store of τ is non-failed. (The proof shows that the same failure would occur sequentially, hence not all rule applications might be required to reach this failure.)

We first show that all parallel rule applications, denoted by elements in R, can be performed sequentially. This is done by an induction over $|R|$ using the following induction hypothesis *(IH) for $k \in \{1, \ldots, |R|\}$:*

There exists $R_k \subseteq R$ with

1. $|R_k| = k$
2. *there exists a multiset $\mathbb{G}'_k$ with $c \in \mathbb{G}'_k \Rightarrow c \in \mathbb{G}$, such that*
   - $\sigma_k = \langle \mathbb{G}'_k; \mathbb{B}; \mathbb{V} \rangle \rightarrowtail^{k}_{\mathcal{P}'} \tau_k = \langle \mathbb{G}_k; \mathbb{B}_k; \mathbb{V} \rangle$ *by applying the rules from R_k*
   - $\left(\bigcup_{(\_,\_,B_c,\_,\_) \in R_k} B_c \right) \subseteq \mathbb{G}_k$
   - $\mathbb{B}_k = \mathbb{B} \wedge \bigwedge_{(\_,\_,\_,B_b,\_) \in R_k} B_b$
   - $\forall c \in \mathbb{G}'_k$ *holds*
     $\#(c, \mathbb{G}'_k) - \#(c, \{c \mid (\_, H_2, \_, \_, \mathbb{B}') \in R_k \wedge (H_2 \wedge \mathbb{B}' \rightarrow c)\}) \in \{0, 1\}$.
3. $\neg\exists(c_1, c_2) \in \mathcal{D}(\sigma, R_k)$ *with* $\exists(c_2, \_) \in \mathcal{D}(\sigma, R \setminus R_k)$.

The first condition corresponds to the number of parallel rules being applied. Condition 2 specifies that the initial state is sufficient to fire all the rules, when extending it with multiples of constraints. It further ensures, that multiples are only added if they are consumed within these k sequential steps. Finally, condition 3 ensures a sequential rule ordering that does not lead to removal of constraints that are required in later rule applications.

Induction over $|R|$:

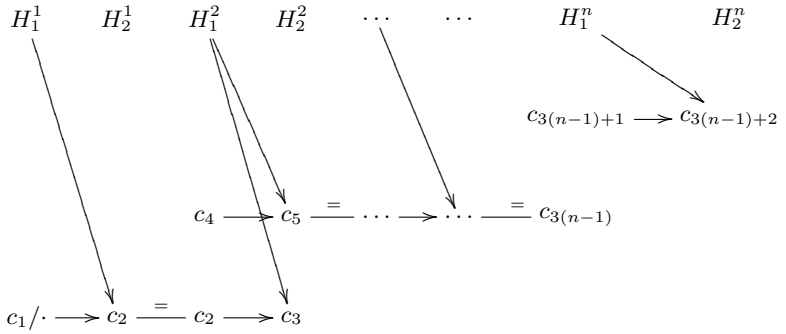
Base case: $|R| = 1 \Rightarrow R' = R = \{(H_1, H_2, B_c, B_b, \mathbb{B}')\}$. *Condition 1 of the IH is trivially satisfied. Condition 2 is clear, as in this case there is only one rule application, coinciding with the sequential case. Condition 3 is satisfied, because $\mathcal{D}(\sigma, R \setminus R_1) = \emptyset$.*

Induction step: *From IH follows that there exists R_k and $\tau_k = \langle \mathbb{G}_k; \mathbb{B}_k; \mathbb{V} \rangle$ as defined above. We need to show: $\exists(H_1, H_2, B_c, B_b, \mathbb{B}') \in R \setminus R_k$ such that $R_{k+1} = R_k \cup \{(H_1, H_2, B_c, B_b, \mathbb{B}')\}$ satisfies conditions 2 and 3 (with 1 being trivially satisfied)*

Condition 2: *Each element of R corresponds to a rule application on the original state σ. Therefore, for each $c \in H_1 \cup H_2$ there exists a unique $c' \in \mathbb{G}$ with $c \wedge \mathbb{B}' \rightarrow c'$. From IH, condition 2, follows that either $\#(c', \mathbb{G}_k) = 1$ or*

$\#(c', \mathbb{G}_k) = 0$. *In the latter case we extend* $\mathbb{G}'_{k+1}$ *by an additional copy of* c'. *Due to monotonicity of CHR we then have* $c' \in \hat{\mathbb{G}}_k$ *(where* $\sigma_{k+1} \rightarrowtail^k \tau'_k = \langle \hat{\mathbb{G}}_k; \mathbb{B}_k; \mathbb{V} \rangle$*). Therefore,* $\mathbb{G}'_{k+1}$ *is constructed such that* $\hat{\mathbb{G}}_k$ *contains all constraints required to match* $H_1 \cup H_2$.
We then have $\tau'_k \rightarrowtail_{\mathcal{P}'} \tau_{k+1} = \langle \mathbb{G}_{k+1}; \mathbb{B}_{k+1}; \mathbb{V} \rangle$ *with the required properties for* $\mathbb{G}_{k+1}$ *and* $\mathbb{B}_{k+1}$ *holding for* R_{k+1}. *The multiplicity property also holds, as either* $d(c) = 1$ *is unchanged, or we added an additional copy that was removed by the* $k+1$*-th rule application resulting in* $d(c) = 0$*: either* $(\_, c') \in \mathcal{D}_k$ *and therefore, by condition 3* $\nexists (c', \_) \in \mathcal{D} \setminus \mathcal{D}_k$, *or this is the first tuple of the form* $(\_, c') \in \mathcal{D}_{k+1} \setminus \mathcal{D}_k$. *Therefore, condition 2 can be satisfied for any element of* $R \setminus R_k$.

Condition 3: *proof by contradiction:*
Assume that there exists no element of $R \setminus R_k$ *that can be added to form* R_{k+1} *such that condition 3 is satisfied. Then* $\mathcal{D}(\sigma, R \setminus R_k)^+$ *is not irreflexive – contradicting deletion-acyclicity – which we prove as follows:*
Assume, that $R_{k+1} = R_k \cup \{(H_1, H_2, B_c, B_b, \mathbb{B}')\}$. *Hence, by assumption there exists* $(c_1, c_2) \in \mathcal{D}(\sigma, R_{k+1})$ *with* $\exists (c_2, c_3) \in \mathcal{D}(\sigma, R \setminus R_{k+1})$ *stemming from another rule application. Therefore,* $\mathcal{D}(\sigma, R \setminus R_k)$ *contains tuples corresponding to elements from* $H_1 \times \{c_2\}$ *(or* $(\cdot, c_2)$ *if* $H_1 = \emptyset$*) and* $\{c_2\} \times H'_2$. *Deletion-acyclicity ensures* c_3 *does not correspond to a constraint in* H_1.
By iteratively selecting each available rule application for R_{k+1} *we find* $n = |R \setminus R_k|$ *constraints* $(c_2, c_5, c_8, \ldots, c_{3(n-1)+2})$ *that occur in the* H_2*-part of one rule and the* H_1*-part of another. W.l.o.g. we ignore* $(\cdot, \_)$ *tuples here, as they only reduce the number* n *of such shared constraints. The cyclicity argument below is unaffected and the assumption that condition 3 cannot be satisfied ensures, that not all tuples are of this form. This situation is depicted below, where* H^i_1, H^i_2 *are the two head parts shown as columns, arrows denote tuples of* $\mathcal{D}(\sigma, R \setminus R_k)$ *and* $=$*-edges mark the* n *constraints from above.*



By the above argumentation $c_{3(n-1)+2}$ *also corresponds to a constraint in some* H_1*-part. Hence, there needs to be an additional equality edge from* $c_{3(n-1)+2}$ *to some previous* $H^i_1 (i < n)$. *However, by Definition 7 we have tuples correspond-*

ing to $H_1^i \times H_2^i \in \mathcal{D}(\sigma, R \setminus R_k)$*. Therefore, any such equality edge causes a cycle to exist in* $\mathcal{D}(\sigma, R \setminus R_k)$*, hence* $\mathcal{D}(\sigma, R \setminus R_k)^+$ *is not irreflexive.*

This proves that all elements from $R \setminus R_k$ *satisfy conditions 1 and 2, and that there exists at least one such element that also satisfies condition 3. Therefore, the required* R_{k+1} *exists and the induction is complete.*

Hence, for R *there exists the computation* $\sigma' \rightarrowtail_{\mathcal{P}'}^{|R|} \tau'$ *according to condition 2. We now compare* τ' *to* τ*: By Definition 6* $\tau = \langle (\mathbb{G} \setminus D) \cup A; \mathbb{B} \wedge B; \mathbb{V} \rangle$*. It is clear from condition 2 that the built-in stores are equivalent. Furthermore, all constraints in* D *have also been removed from* $\mathbb{G}'$ *(as the same rules have been applied and condition 2 holds).*

Therefore, the only difference between τ *and* τ' *is that constraints added by the rule applications could occur in multiplicities greater than* 1 *(whereas* A *is a set). Hence, we can make use of the set-semantics rules to reduce multiplicities to* 1 *resulting in* $\tau' \rightarrowtail^* \tau$*.*