Fine-grained Parallel Implementation of the Preflow-Push Algorithm in CHR

Marc Meister

Fakultät für Informatik, Universität Ulm, Germany
Marc.Meister@uni-ulm.de

Abstract. Constraint Handling Rules (CHR) is a concurrent, committed-choice, rule-based language. Recently it was shown that programs for standard (operational) semantics can be interpreted in a parallel computation model. As case study, the classical, imperative, parallel, and non-confluent preflow-push algorithm is implemented in CHR for refined and for standard semantics.

1 Introduction

Constraint Handling Rules (CHR) [5] is a concurrent, committed-choice, rule-based language which was originally developed for rapid prototyping and for writing constraint solvers. The *operational* semantics of a CHR program is given by a state transition system and comes in different flavours.

In *refined* (operational) semantics, the constraints in a state are evaluated from left-to-right and rules are tried in textual program order. Concrete implementations, like SICStus Prolog with CHR [14], use this refined semantics to execute rules *sequentially.* To the contrary, the more abstract *standard* (operational) semantics makes no assumptions about the order constraints are evaluated and rules are tried. As *constraint removal* caused by the application of a rule remains atomic, standard semantics can be interpreted as a *parallel* computation model [6]. A CHR runtime system for standard semantics can simulate a parallel random access machine (PRAM) [8], where copies of the rules make up the asynchronous threads, and rules operate on a global constraint store.

As refined semantics is an instance of the standard semantics [4], any correct program for standard semantics is also correct for refined semantics. However, only *confluent* programs have this property for the reverse direction. In general, a correct program for refined semantics is not correct in standard semantics: The CHR program (using SICStus Prolog syntax)

```
a \ b <=> write('yes').
    b <=> write('no').
```

can be used to test if constraint `a` is present in refined semantics. In standard semantics, however, the answer for the goal `b` is `no` even if `a` is present, if the second rule is chosen for execution.

Recently, classical algorithms – e.g., the classical union-find algorithm – have been implemented in CHR for refined [13] and for standard semantics [6]. We follow this trend to use CHR as a general-purpose programming language by investigating CHR implementations of the *preflow-push* algorithm for both semantics.

The preflow-push algorithm [7] solves the maximal flow problem: Intuitively the problem can be visualised as a system of connected water-pipes, where each pipe has a given size restricting its capacity. The system is closed, s.t. no water can escape, except for one source and one sink valve. Solving the maximal flow problem accounts to finding the maximal capacity the system can handle from source to sink (and to find the routes the water actually takes). The preflow-push algorithm consists of two local actions that are applied exhaustively. Applications for finding a maximal flow are manifold and found in, e.g., transportation planning and resource management. In constraint programming, the maximal flow is needed for the efficient handling of the global alldifferent and global cardinality constraints [10, 15].

We implement the (inherently non-confluent) preflow-push algorithm, both for refined and for standard semantics. The transformation from refined to standard semantics showed several problems when implementing a classical, parallel, imperative, and non-confluent algorithm in parallel CHR. We wanted to allow fine-grained parallel execution on disjoint parts of the graph for a high level of parallelism.

We address problems for programming in parallel CHR. Our solutions are implemented and tested: These include a basic `for`-loop control structure, concurrent locking without global control, and some basic results how to assemble a CHR programs using parallel CHR subprograms. In our case study, we encode sequential blocks of the parallel preflow-push algorithm by chains of `for`-loops.

Parallel, imperative implementations [1] are not comparable to our work due to fundamentally different computing models. Closest to our work is [6]: Frühwirth uses confluence analysis to prove that a given CHR program is correct in standard semantics. In our approach, however, we split the given (con-confluent) problem into subproblems, which have a standard semantics implementation.

This paper is organised as follows: We briefly introduce the preflow-push algorithm in Sect. 2 and the connection between standard operational semantics and parallelism in Sect. 3. We discuss problems when making the move from refined to standard semantics and propose a loop abstraction in Sect. 4. We present our versions of the preflow-push algorithm in CHR both for refined and for standard semantics in Sect. 5 and then conclude.

2 Generic Preflow-Push Algorithm

We recall the necessary notation from graph theory before explaining the generic preflow-push algorithm [7] (see, e.g., [3] for a detailed introduction).

A *flow network* is a complete, directed graph $G = (V, E)$, with vertices V and edges E, where each edge $(u, v) \in E = V \times V$ is assigned a non-negative capacity $c(u, v)$. A flow network contains the two distinguished vertices `source` and `sink`. A *flow* in a flow network is a function $f : E \to \mathbf{R}$ obeying

– capacity restriction $\forall u, v \in V : f(u, v) \leq c(u, v)$,
– skew symmetry $\forall u, v \in V : f(u, v) = -f(v, u)$, and finally
– flow conservation $\forall u \in V \setminus \{\texttt{source}, \texttt{sink}\} : \sum_{v \in V} f(u, v) = 0$.

A solution to the *maximum-flow problem* is given by a valid flow which maximises the flow value $\sum_{u \in V} f(u, \texttt{sink})$ (which is equal to $\sum_{u \in V} f(\texttt{source}, u)$) of a flow network. Usually, there are several valid flows for a flow value.

Instead of the augmentation path (Ford-Fulkerson) algorithms which examine the complete network, we focus on the more localised preflow-push algorithms, which relax the flow conservation property and allow vertices to overflow during the computation: Vertices may overflow during the execution of the algorithm, i.e., for all $u \in V$ the *excess flow* $e(u) = \sum_{v \in V} f(v, u)$ can be positive. However, upon termination for all $u \in V \setminus \{\texttt{source}, \texttt{sink}\}$, the excess flow must be zero in order to make the *preflow* f a valid flow.

The preflow-push algorithms employ a global label height $h : V \to \mathbf{N}$ and the actions *push* and *lift*, which are applied in arbitrary order – hence “generic” preflow-push algorithm. The general preflow-push algorithm is given in Fig. 1. When no action is applicable any more, the preflow f is a valid flow with maximal flow value.

For every edge $(u, v) \in E$ the *residual capacity* $r(u, v)$ is defined by $c(u, v) - f(u, v)$ where f is a preflow. We call an edge $(u, v) \in E$

– a *flow edge* if $f(u, v) > 0$,
– a *residual edge* if $r(u, v) > 0$, and finally
– a *downward residual edge* if $r(u, v) > 0$ and $h(u) > h(v)$.

The preflow-push algorithm has two important properties: The vertex heights never decrease throughout the computation and for all *residual edges* (u, v) the invariant $h(u) \leq h(v) + 1$ holds.

Initialisation:

- $f \leftarrow 0$, except for $f(\texttt{source}, u) \leftarrow c(\texttt{source}, u)$ $(u \in V)$.
- $h \leftarrow 0$, except for $h(\texttt{source}) \leftarrow \#V - 2$.
- $e \leftarrow 0$, except for $e(u) \leftarrow c(\texttt{source}, u)$ $(u \in V)$.

Apply exhaustively (until no more change):

"push(u,v)" applies when $e(u) > 0$, $r(u, v) > 0$, and $h(u) > h(v)$.
Then do push $p \leftarrow \min\{e(u), r(u, v)\}$ units of flow from u to v. Update by subtracting p from $e(u)$, $f(v, u)$, and $r(u, v)$ while adding p to $e(v)$, $f(u, v)$, and $r(v, u)$.

"lift(u)" applies when $e(u) > 0$, $\forall v \in V : r(u, v) > 0 \Rightarrow h(u) \leq h(v)$, and $u \neq \texttt{source}$, $u \neq \texttt{sink}$.
Then do lift vertex u to height $h(u) \leftarrow 1 + \min\{h(v) : r(u, v) > 0, v \in V\}$.

Fig. 1. Generic preflow-push algorithm

Simultaneous actions of the preflow-push algorithm, working on *disjoint* parts of the flow graph, allow for *parallel execution*. The preflow-push algorithm is therefore *inherently parallel*. However, each *push* or *lift* action consists of a *sequential program*, e.g., compute the minimum height *before* updating the height. For integral capacities the maximal parallelism is bounded by the sum of possible excess flow. Parallelisation can worsen the performance: Consider a flow graph with exactly one path from `source` to `sink`.

3 Constraint Handling Rules (CHR)

Constraint Handling Rules (CHR) [5] is a concurrent, committed-choice, rule-based language. We assume basic familiarity about CHR (for an introduction visit the CHR web page [12]).

3.1 Standard and Refined Operational Semantics

The *standard* operational semantics of CHR is given by a transition system. For a query (a conjunction of user and built-in constraints) rules are applied until a fix-point is reached. For each kind of CHR rule, a transition is given: A simplification rule $H \Leftrightarrow G \mid B$ can apply in state $(H' \wedge C)$, if built-in constraints C_b of C entail that H' matches the head H and the guard (Fig. 2). A simplification rule *replaces* instances of CHR constraints. A simplification rule to avoid non-termination.

Any rule that is applicable can be applied and rule application cannot be undone (because CHR is a committed-choice language). Standard semantics allows *unfair* rule application.

The *refined* (operational) semantics is an instance of the *standard* semantics [4], curbing non-determinism. In refined semantics, the constraints in a state are evaluated from left-to-right, rules are tried in textual program order, and *constraint insertion* (of body constraints of executed rules) is left-to-right. Hence, any correct program for standard semantics is also correct for refined semantics. However only *confluent* programs have this property for the reverse direction. In general, a correct program for refined semantics is not correct in standard semantics.

Orthogonal to the aspect of the operational semantics is the aspect of the *declarative* semantics of a CHR program, which is not the topic of this paper.

3.2 Parallelisation for free in Standard Semantic

Strong parallelism of CHR [6] formalises that rule applications may share constraints if they leave these constraints unchanged, c.f. Fig. 3 where B_i,H_i, and C are conjunctions of constraints. We assume that *constraint removal* in standard semantics, caused by a rule application, is *atomic*: If two rules are applicable to remove a given constraint, then only one can succeed. Opposed to the

$$\begin{array}{ll} \text{IF} & H \Leftrightarrow G \mid B \text{ is a fresh variant of a rule with variables } \bar{x} \\ \text{AND} & CT \models \forall (C_b \rightarrow \exists \bar{x}(H = H' \wedge G)) \\ \text{THEN} & (H' \wedge C) \mapsto (B \wedge G \wedge H = H' \wedge C) \end{array}$$

Fig. 2. Simplify state transition computation

$$\begin{array}{lll} \text{IF} & H_1 \wedge C & \mapsto B_1 \wedge C \\ \text{AND} & H_2 \wedge C & \mapsto B_2 \wedge C \\ \text{THEN} & H_1 \wedge H_2 \wedge C & \mapsto B_1 \wedge B_2 \wedge C \end{array}$$

Fig. 3. Strong parallelism of CHR

refined semantics, we impose no order on the *constraint insertion*, caused by a rule application. This means, that the insertion of a sequence of constraints can happen in any order (e.g., all at once) and is not due until all other computation quiesces.

Clearly, any program for standard semantics *can* enjoy strong parallelism. We emphasise can, as the level of parallelism depends on the actual problem representation and program rules. A CHR program enjoys fine-grained parallel execution only, if the actual implementation allows rules to remove (disjoint sets of) constraints simultaneously. Encoding the complete problem into a single constraint voids the effort of any parallelism.

Refined operation semantics can be interpreted as a *sequential* execution model, while, on the other hand, standard operational semantics can be identified with a *parallel* execution model. Available compilers (like SICStus Prolog with CHR [14]) use the refined semantics. Standard semantics can be *simulated* in a CHR program for refined semantics as an *interleaving semantics.*

4 Programming in Standard Semantics

In this section we explore implications for implementing classical imperative algorithms in CHR for standard semantics, i.e., what is programming in *parallel CHR* like.

4.1 From Refined to Standard Semantics

Any confluent CHR program automatically enjoys strongly parallelism. So taking the effort to prove a program to be confluent pays off. Performing a confluence analysis, the programmer has to keep an eye on the problem structure in order to actually allow parallelism, i.e., allow multiple (copies) of rules to work on shared constraints. With the necessary insight, this approach proved successful for the union-find algorithm [6].

Usually, CHR programs implemented as prototypes rely on the refined semantics, e.g., on the order rules are tried. Often, a rule is deliberately put at the end of the program, as it must not apply if some rule (which comes *before* in textual program order) is applicable. However, if the implemented algorithm itself turns out to be inherently non-confluent, like e.g., the preflow-push algorithm which returns only *a* valid maximal flow, a confluent and fine-grained parallel implementation is not obvious.

In this paper, we split the classical, imperative algorithm in subproblems – where each subproblem can be implemented in standard semantics – and compose the resulting subproblems. This compossible programming style easily fits with imperative programs (which are usually composed from a number of basic operations). Therefore our general goal in implementing problems in CHR for standard/parallel semantics is *re-usability*: A CHR program for standard semantics can serve as subprogram in another (not necessarily parallel) program.

4.2 Problems and Solution with Standard Semantics

We propose to categorise CHR constraints according to their behaviour in terms of their life-span, whether they might be updated by changing their parameters, and by the number of copies in the store during execution. This showed to be convenient when reasoning about CHR programs.

Backbone constraints are unique (there is only a single copy) and remain unchanged in the constraint store during execution.

Info constraints map fixed key-arguments to the remaining value-arguments (we assume keys to be unique). An updating rule that removes an info constraint reinserts it in its body (unconditionally).

Flag constraints are created and removed during runtime and describe integral values by the corresponding number of copies in the store.

By this "separation of concerns", a rule which, e.g., removes an info constraint without reinserting it again, is easily marked as possible programming error.

We encountered the following main problems, when moving from refined to standard semantics. For each problem we propose a solution.

(P1) The constraint store can no longer be seen as a query-able data-structure. We cannot detect the *absence* of the CHR constraint. As remedy we propose to explicity store the absence of a property as a constraint (introductory example, `ok/1` constraint in Tab. 1).

(P2) If a rule should be tried for application when a constraint *becomes absent*, we propose to trail the number of copies of the constraint that are in the store. Trailing is done by *info*-constraints and the corresponding info-constraints are added to the head of the rule (`down/2` constraint in Tab. 1).

(P3) Concurrent activity can change the store during the execution of *several* rules in ways which are hardly predictable: Counting the number of constraints (of some type) which are in the store might result in too large numbers, as some of the constraints are changed by concurrent activity. As remedy we introduce *backbone*-constraints which remain (unchanged) during the execution and operate on copies of them (Ex. 1).

(P4) In order to implement a sequential (sub)problem in CHR for parallel semantics we propose to use a set of *transition rules* to move on from one phase to the next. In our case, a *phase* consists of an instance of a `for`-loop (described in the following section), which is started by posting a `for/3` constraint and the successful completion is indicated by the presence of the corresponding `done/3` constraint. A transition rule simply removes the answer constraints of one phase and posts constraints creating the following phase. This allows to encode explicit control flow for sequential parts in standard semantics easily (Fig. 7).

Usually, backbone constraints represent the (fixed) underlying problem structure (P3), info constraints take program variables (to allow for non-logical updates) (P2), and flag constraints are used mainly to guide the control flow (P1).

4.3 A `for`-Loop Control Abstraction for Standard Semantics

As CHR lacks even basic control structures (known from imperative programming) and to promote general purpose programming in CHR, we propose a basic `for`-loop control abstraction composed of the CHR rules in Fig. 4.

```
for_zero  @ for(Id,0,Info)                        <=> done(Id,0,Info).
for_intro @ for(Id,S,_)                           ==> S > 0 | id(Id).
for_step  @ inc(Id)                               <=> count(Id,1).
for_count @ count(Id,A), count(Id,B)              <=> C is A+B, count(Id,C).
for_done  @ id(Id), for(Id,N,Info), count(Id,N) <=> done(Id,N,Info).
```

Fig. 4. `for`-loop rules (standard semantics)

A `for`-loop which is to iterate *at most* $n \geq 0$ times over a user-provided loop-body is initialised by posting the flag `for(Id,`n`,Info)` where, the first argument is an unique identifier `Id`. During the life-span of the loop, the flag `id(Id)` is present and rules for the loop step – provided by the user – post exactly n copies of the flag `inc(Id)`, where each copy of the flag `inc(Id)` indicates that one loop step is finished. The flag `done(Id,`n`,Info)` (as the only remaining constraint from the abstraction) indicates that the loop is finished. The loop abstraction interacts with other rules via the constraints `for`, `id`, `inc`, and `done` as demonstrated in Ex. 1.

Note that a loop is defined by constraints (and not tied to a specific place in the source code as in an imperative languages). While the rules in the loop abstraction increment the loop variable, the user-provided rules take care of the loop initialisation and the loop body as shown in the following examples.

Example 1. We calculate the minimum of all $n > 0$ program variables. A program variable with name x and value $v(x)$ is stored in a backbone constraint `cname(`x`)` and an info constraint `cdata(`x`,`$v(x)$`)` (these are considered to be the only constraints in the store). As concurrent activity might change values during the life-span of the loop, we make a copy of the backbones in rule `e2` and the flags `pie(Id-min,`x`)` are consumed ("eaten") for retrieving the values $v(x)$ in rule `e3` (Problem P3). Loop initialisation is done by rules `e1` and `e2` and the loop step rules consist of rules `e3` and `e4`.

```
e1 @ id(Id-min)                    ==> min(Id-min,+inf).
e2 @ id(Id-min), cname(X)          ==> pie(Id-min,X).
e3 @ cdata(X,V)\ pie(Id-min,X)     <=> min(Id-min,V).
e4 @ min(Id-min,A)\ min(Id-min,B) <=> A =< B | inc(Id).
```

Calculation is started by posting the flag `for(Id-min,`n`,'')` and the minimum m is claimed by removing `done(Id-min,_,_), min(Id-min,`m`)`. Altogether $n+1$ copies of the flag `min/2` are generated and after n applications of rule `e4` the loop is finished.

Example 2. Similarly, the following three rules allow to sum up the values.

```
a1 @ id(Id-add), cname(X)          ==> pie(Id-add,X).
a2 @ cdata(X,V)\ pie(Id-add,X)     <=> add(Id-add,V).
a3 @ add(Id-add,A), add(Id-add,B) <=> C is A+B, add(Id-add,C), inc(Id).
```

The third parameter of the `for/3` and `done/3` constraints can be used when several `for`-loops are chained by transition rules and some information needs to passed along (c.f. Fig. 7). We re-use both the minimum loop and the addition loop as confluent subroutines in our preflow-push algorithm for standard semantics.

5 Preflow-Push in CHR

In this section we present our implementations of the preflow-push algorithm in CHR. While the refined/sequential version is straightforward, the standard/parallel version is somewhat involved (with only parts of it shown, due to lack of space). However the complete source code is available [9].

5.1 Modelling the Flow Problem

We restrict ourselves to unit capacities $c(u,v) \in \{0,1\}$ (these are needed for the bipartite matching underlying the alldifferent implementation) and assume (by prepossessing and adding up capacities between same vertices) that $c(u,v) + c(v,u) \leq 1$ holds for all $u,v \in V$. Positive excess flow is stored as $e(u)$ copies of the flag constraint `e(`u`)`. Nevertheless these (apparent) restrictions can easily be overcome be storing capacity and excess flow in info constraints. Unit capacities (and only these are needed for bipartite matching in an alldifferent implementation) make each edge (u,v) with $c(u,v) = 1$ or $c(v,u) = 1$ either a residual edge $r(u,v) = 1 \wedge f(u,v) = 0$ or a flow edge $f(u,v) = 1 \wedge r(u,v) = 0$ (therefore we can skip the minimum computation in the *push* action, c.f. Fig. 1).

Thus, we can used the following representation of the current preflow f which is updated repeatedly during run-time: We *only* consider edges (u,v) with capacity $c(u,v) = 1$ and for every such edge, we use a flag constraints `res/2` to discriminate flow edges from residual edges:

– `res(`u`,`v`)` iff (u, v) is a residual edge, and
– `res(`v`,`u`)` iff (u, v) is a flow edge.

By pushing flow along the residual edge (u, v), the flag `res(`u`,`v`)` is replaced by `res(`v`,`u`)`.

To model the flow problem and to record the current state of the preflow-push algorithm, we use the constraints given in Tab. 1.

Table 1. Modelling the preflow-push algorithm in CHR constraints

Constraint	Category	Description
`h(`u`,`$h(u)$`)`	info	height $h(u)$ of vertex u
`e(`u`)`	flag	(one unit of) excess flow in vertex u
`res(`u`,`v`)`	flag	residual edge (u, v)
`cap(`u`,`v`)`	backbone	$c(u, v) > 0$
`vertex(`u`,`$n(u)$`)`	backbone	$n(u) = \#\{v \in V : c(u, v) > 0 \vee c(v, u) > 0\}$
`down(`u`,`$d(u)$`)`	info	$d(u) \geq \#\{v \in V : r(u, v) > 0, h(u) > h(v)\}$
`lock(`u`,`v`,`$l(u, v)$`)`	info	edge lock $l(u, v) \in \{u, v, \texttt{clear}\}$
`ok(`u`)`	flag	vertex u is not lifting

5.2 Preflow-push in Refined Semantics

The key insight for the concise implementation (Fig. 5) of the preflow-push algorithm for refined semantics is that *vertex u (neither* `source` *nor* `sink`*) can lift iff no push is possible from u.*

This global coordination between *push* actions applied exhaustively, followed by exactly one *lift* action, and returning for further *push* actions is implemented by a flag constraint `phase(`*Phase*`)`. *Phase* is either `push`, `lift`, or `update`: Rule `r_push` encodes a *push* and the rules `r_lift1`, `r_mcalc`, `r_minit`, and `r_lift2` encode a *lift* action. As long as *push* actions apply (in phase `push`), flow is pushed downwards, replacing a residual edge with a residual edge in the opposed direction. By temporarily removing `phase(push)` in rule `r_push`, we prohibit possible overlapping instances of rule `r_push`. Rule `r_trans` applies when no *push* is possible (as rule `r_push` comes *before* in textual order). In phase `lift`, either rule `r_lift1` applies when a *lift* action is possible, or the computation terminates with a valid maximal flow. In phase `update`, the minimum height of the residual neighbours is calculated by the rules `r_mcalc` and `r_minit`. Again we exploit the refined semantics (as rules `r_mcalc` and `r_minit` come *before* in textual program order and are therefore no longer applicable). Rule `r_lift2` then updates to the new height and returns `phase(push)` to allow for further *push* actions.

```
r_push  @ h(U,UH), h(V,VH)\ e(U), res(U,V), phase(push) <=>
          UH > VH | res(V,U), e(V), phase(push).

r_trans @ phase(push) <=> phase(lift).

r_lift1 @ e(U)\ h(U,HU), phase(lift) <=> U\=source, U\=sink | getmin(U), phase(update).
r_mcalc @ minimum(A)\ minimum(B) <=> A =< B | true.
r_minit @ getmin(U), res(U,V), h(V,H) ==> minimum(H).
r_lift2 @ getmin(U), minimum(M), phase(update) <=> M1 is M+1, h(U,M1), phase(push).
```

Fig. 5. Preflow-push algorithm (refined semantics)

In our sequential implementation of the preflow-push algorithm in CHR we rely on the fixed order of constraint visits and rule tries in refined semantics for means of control.

Example 3. To calculate the maximum flow for the simple flow graph $V = \{\texttt{source}, \texttt{sink}, \texttt{x}\}$ with positive capacities $c(\texttt{source}, \texttt{x}) = c(\texttt{x}, \texttt{sink}) = 1$, note how the initialisation and the representation is encoded in the CHR query.

```
?- res(x,source), res(x,sink), h(source,1), h(x,0), h(sink,0), e(x), phase(push).
res(x,source), res(sink,x), h(source,1), h(x,1), h(sink,0), e(sink), phase(lift)?
```

By tracking back residual edges in the answer – starting from the `sink` – we find the maximal flow: From `source` via vertex `x` (lifted to height $h(\texttt{x}) = 1$) to `sink` with maximal flow value one.

5.3 From Refined to Standard Semantics

Our challenge is: Implement the *push* and *lift* actions which a high-level of parallelism. To this end, guarantee that parallel *push* actions do not interfere with *lift* actions, and parallel lifting only applies at *disjoint* parts of the flow graph. While a *push* can be encoded as a single rule (thus enjoying the atomic removal property) this is not the case for the *lift* action. Lifting a vertex requires a *sequential program* by calculating the new height *before* actually lifting. As standard semantics provides no means to control atomicity spanning the execution of *several* rules, precautions have to be taken that parallel activity does not interfere with lifting. To disallow any two neighbouring vertices from entering the critical phase while lifting at the same time we implemented a locking mechanism.

We encode the necessary sequential (sub)program by *phase transition rules.* While the main rules (Sect. 5.4) of our implementation for parallel semantics are easy, concurrent locking (Sec. 5.5) and destructive updates are rather tedious.

5.4 Push and Lift in Standard Semantics

The `s_push` and `s_lift` rules for the standard semantics in Fig. 6 resemble closely the *push* and *lift* actions from Fig. 1. The presence of the flag constraint `ok(`u`)` indicates that vertex u is *not lifting* and therefore, u may push and receive flow (Problem P1). The info constraint `down(`u`,`$d(u)$`)` trails an upper bound of the number of downward residual edges $d(u)$. When `down(`u`,0)` becomes present, *no* downward residual edges are available and we can check, if vertex u is liftable (Problem P2). The backbone constraint `vertex/2` stores the number of neighbours for a vertex in the graph (Problem P3). Initially, all vertices have zero downward residual edges and are endowed with the flag `ok/1`.

```
s_push @ h(U,UH), h(V,VH), ok(U), ok(V)\ res(U,V), e(U), down(U,UM)
         <=> UH > VH | UM1 is UM-1, res(V,U), e(V), down(U,UM1).

s_lift @ e(U), vertex(U,N), down(U,0)\ ok(U)
         <=> U\=source, U\=sink | add(U-try,0), for(U-try,N,'').
```

Fig. 6. Rules `s_push` and `s_lift` (standard semantics)

When pushing flow from vertex u, the number of downward residual edges $d(u)$ is decremented in rule `s_push`. When `s_lift` executes for vertex u, the flag `ok(`u`)` is removed to prevent u from participating in (concurrent) *push* actions and from multiple lifting. Vertices with no `ok/1` flag are invisible to the rules of Fig. 6.

While (copies of) the rules `s_push` and `s_lift` can apply in parallel, the actual lifting (initialised by an application of `s_lift`) could not be encoded in a single rule as the new height $h(u)$ needs to be computed *before* the affected `down/1` constraints can be updated.

During lifting vertex u, neighbours must not increase their height, as the `down/2` constraint of u must be updated and this can be done only *after* u increased its height; the limbo in between must be protected from parallel inference. This is done by requiring vertex u to be locked (and preventing any two neighbours to lock at the same time) before entering these critical parts of lifting. Locking is initialised by posting (add(u-try,0), for(u-try,$n(u)$,'')) by rule s_lift and discussed in the following section. After vertex u locks successfully in the minimum height is calculated in a phase u-min, reusing the code from Example 1 before a transition rule moves on to phase u-update where affected down/2 constraints are updated. Finally, the lock is released in phase u-unlock and ok(u) is restored. *Correctness of Implementation:r* The rules s_push and s_lift (Fig. 6) correspond to the generic *push* and *lift* actions. Therefore, we can rely on the termination and correctness properties of the generic preflow-push algorithm:

- When the flags ok(u) and ok(v) are present, the height-invariant along the residual edge (u, v) holds. Therefore s_push operates on a valid preflow.
- When u has no downward residual edges and ok(u) is present, eventually down(u,0) is present, making the necessary condition for application of s_lift available.
- A vertex u, selected by rule s_lift for lifting, eventually succeeds to lock.
- A flag ok(u), removed by the s_lift-rule, is eventually restored after u finished lifting. We have to show that the phases u-min, u-update, and u-unlock do not depend on the phases of other vertices and terminate eventually.

5.5 Locking in Standard Semantics

Compared to refined semantics, where locking was not an issue due to sequential execution, locking in standard semantics is an issue due to parallel execution. By locking vertices, we temporarily rule out some computations, i.e., we disallow neighbouring vertices to be both in the critical phase during lifting. We solved the problem of *concurrent locking without global control* by building a dependency graph recording failed locking attempts. To this end we *re-used* the for-loop abstraction repeatedly and encoded sequential control decision as transition rules. While tricky, the implementation is necessary as CHR does not offer higher-level synchronisation features.

```
t0 @ id(U-try), cap(U,V)            ==> pie(U-try,U-V).
t1 @ id(U-try), cap(V,U)            ==> pie(U-try,V-U).
t2 @ lock(A,B,clear), pie(U-try,A-B) <=> lock(A,B,U), add(U-try,1).
t3 @ lock(A,B,V)\     pie(U-try,A-B) <=> V \= clear, U \= V | pie(U-retry,V), inc(U-try).
t4 @ done(U-try,N,_), add(U-try,S)   <=> (S=N -> for(U-min,N,'') ; for(U-cancel,S,N)).

c1 @ id(U-cancel)\ lock(A,B,U)       <=> lock(A,B,clear), inc(U-cancel).
c2 @ done(U-cancel,S,N)              <=> R is N-S, for(U-retry,R,N).

r1 @ ok(V)\ pie(U-retry,V)           <=> inc(U-retry).
r2 @ id(V-retry)\ pie(U-retry,V)     <=> V@<U | inc(U-retry).
r3 @ done(U-retry,_,N)               <=> add(U-try,0), for(U-try,N,'').
```

Fig. 7. Locking rules (standard semantics)

Our locking mechanism consists of the phases try, cancel, and retry: If vertex u could not successfully claim all edges in u-try, then, after some cleanup in phase u-cancel and after all dependencies are cleared in u-retry, it returns to phase u-try for another try (Fig. 7). For each backbone capacity constraint cap(u,v), the info constraint lock(u,v,$l(u,v)$) indicates (in the third argument) if (u, v) is claimed by u, claimed by v, or is clear. Vertex u is *locked* when it can claim all edges (u, v), thus neighbouring vertices cannot lock at the same time.

6 Conclusion

We follow the trend to use CHR as a general purpose programming language. We implemented the preflow-push algorithm in CHR both for the refined (sequential) and the standard (parallel) semantics, tailored to allow fine-grained parallel execution. Implementing a well studied classical algorithm in parallel CHR is by no means easy, as CHR lacks basic control features like loops. Problems encountered for programming in standard semantics were addressed. A parallel loop-control abstraction and a fine-grained locking mechanism were proposed, implemented, and successfully used. Our quest for a hight level of parallelism turned out to be somewhat involved – altogether the find-grained parallel implementation required 29 rules as opposed to the six rules in refined semantics.

In this exploratory work we gained some insight into how to implement a classical, parallel, imperative, and non-confluent algorithm in parallel CHR. We found that re-usability of the parallel loop-control abstraction actually pays off.

The achieved parallelism can (up to now) not be reflected in actual speedup, as no appropriate compiler/hardware for the standard semantics is available. When extending CHR with control structures for standard semantics, our loop abstraction and locking mechanism may prove useful. The lack of syntactic sugar, which makes the expression of loops verbose, has already been noted for multiset rewriting in GAMMA [2] and motivated a loop abstraction for Prolog [11].

Future work should explore the simulation of standard semantics as interleaving semantics in refined semantics. Also, the semaphore-like locking mechanism needs further investigation.

Regarding the preflow-push algorithm, termination, correctness and complexity (how much parallelism is achieved) need to be investigated. Future work should extend the implementation both with the gap heuristic and with the periodic global relabelling heuristic [1] in CHR. A long-term goal is the efficient and parallel implementation of Régin's alldifferent constraint in CHR, which uses maximal matching (a special instance of the maximal-flow problem).

References

1. Richard J. Anderson and João C. Setubal. On the parallel implementation of goldberg's maximum flow algorithm. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 168–177, 1992.
2. Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *Commun. ACM*, 36(1):98–111, 1993.
3. Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.
4. Gregory J. Duck, Peter J. Stuckey, María J. García de la Banda, and Christian Holzbaur. The refined operational semantics of constraint handling rules. In *ICLP*, volume 3132 of *LNCS*, pages 90–104. Springer, 2004.
5. Thom Frühwirth. Theory and practice of constraint handling rules. *J. Log. Program.*, 37(1-3):95–138, 1998.
6. Thom Frühwirth. Parallelizing union-find in constraint handling rules using confluence. In M. Gabbrielli and Gupta G., editors, *Logic Programming: 21st International Conference, ICLP 2005*, volume 3668 of *Lecture Notes in Computer Science*, pages 113–127. Springer-Verlag, October 2005.
7. Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.
8. Michael T. Goodrich. Parallel algorithms column 1: models of computation. *SIGACT News*, 24(4):16–21, 1993.
9. Marc Meister, Nov 2005. `http://www.informatik.uni-ulm.de/pm/index.php?id=126`.
10. Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI*, pages 362–367, 1994.
11. Joachim Schimpf. Logical loops. In Peter J. Stuckey, editor, *ICLP*, volume 2401 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 2002.
12. T. Schrijvers and T. Frühwirth. The Constraint Handling Rules (CHR) webpage, Nov 2005. `http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/`.
13. Tom Schrijvers and Thom Frühwirth. Analysing the chr implementation of union-find. In Armin Wolf, Thom Frühwirth, and Marc Meister, editors, *W(C)LP*, volume 2005-01 of *Ulmer Informatik-Berichte*, pages 135–146. Universität Ulm, Germany, 2005.
14. SICStus Prolog Homepage, Dec 2004. SICStus 3.11.2, `http://www.sics.se/sicstus`.
15. W.J. van Hoeve. The alldifferent constraint: A survey. In K.R. Apt, R. Bartak, E. Monfroy, and F. Rossi, editors, *Proceedings of the 2001 ERCIM Workshop on Constraints*, Prague, 2001.