Extending CHR with Negation as Absence

Peter Van Weert, Jon Sneyers⋆, Tom Schrijvers⋆⋆, Bart Demoen

Dept. of Computer Science, K.U.Leuven, Belgium
`{petervw,jon,toms,bmd}@cs.kuleuven.be`

Abstract. In this exploratory paper we introduce $\mathrm{CHR}^{\neg}$, an extension of the CHR language with negation as absence, an established feature in production rule systems. Negation as absence is a procedural notion that allows a more concise and clean programming style. We propose a formal operational semantics for $\mathrm{CHR}^{\neg}$ close to CHR's refined operational semantics. We illustrate and motivate its properties with examples.

1 Introduction

Constraint Handling Rules (CHR) [11, 17] is a high-level programming language extension based on multi-headed committed-choice rules. CHR is constraint-driven: *adding* constraints can cause rules to fire, depending on the *presence* of other constraints. The *removal* and *absence* of constraints has never received much attention in the past. This asymmetry stems from the original intended use of CHR: high-level and declarative prototyping of constraint solvers. In this context, constraints are not *removed* — they are merely *replaced* by equivalent and simpler constraints (hence the term "simplification rule").

However, CHR is increasingly used as a general programming language, in a wide range of applications [12, 18]. As a result, 'constraints' often correspond to elements in some data structure, operations that inspect or modify these elements, auxiliary programming constructs, flags, locks, loops, *etc.* Often, the absence of such 'constraints' is meaningful. Many CHR programmers are tempted to introduce auxiliary constraints and/or rules to test for the absence of constraints. Even worse, triggering rules on constraint removal requires numerous cross-cutting changes. Clearly, such ad-hoc solutions are very cumbersome and error-prone, and lead to more verbose and less declarative programs.

Closely related to CHR are *production rules* [9, 10, 15] (or *business rules*, as they are fashionably called now). These rule languages traditionally [8, 16] offer *negation as absence*. In this paper we introduce and explore an extension of CHR, called $\mathrm{CHR}^{\neg}$, that allows negated heads in the left-hand side of rules. In the production rules literature, and in the rest of this paper, the words "negation", "negated", and "negatively" and the symbol "¬" (pronounced *not*) are often used for negation *as absence*, even though negation as absence is not related to the classical logical negation (at least not in an obvious way).

⋆ Research funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

⋆⋆ Research Assistant of the Research Foundation - Flanders (F.W.O.-Vlaanderen).

Overview. In Section 2 we start by introducing CHR$^{\neg}$ using a series of clarifying examples. Next, Section 3 defines the formal operational semantics of CHR$^{\neg}$, followed by a critical discussion in Section 4. Finally, Section 5 concludes.
An extended version of this paper is available as a technical report [20].

2 CHR$^{\neg}$ by Example

In this section we introduce CHR$^{\neg}$ by means of several small examples. The examples are valid under the refined operational semantics $\omega_r^{\neg}$ of CHR$^{\neg}$, which we describe informally in this section and formally in the next one. For now, it suffices to know that it is based on the conventional operational semantics ω_r of CHR [6]. First we extend the regular CHR syntax [11, 13] with negated heads.

2.1 Syntax and terminology

The general form of a CHR$^{\neg}$ rule is:

$$\mathit{name} \;@\; H_1 \setminus H_2 \setminus\!\!\setminus N_1 \mid G_1 \setminus\!\!\setminus \ldots \setminus\!\!\setminus N_k \mid G_k \Longleftrightarrow G \mid B \tag{1}$$

The *heads* (H_1, H_2, N_1, ..., N_k) are conjunctions of CHR constraints. The heads inherited from regular CHR — H_1 (the *kept* constraints) and H_2 (the *removed* constraints) — are referred to as *positive heads*. Depending on the context, we also use the term *positive head* to denote $H_1 \wedge H_2$. The N_i (for $i \in \{1, \ldots, k\}$) are *negated heads*. If $k = 0$ the rule has no negated heads. Each negated head N_i has a *negated guard* G_i. Negated heads are not allowed to be empty. Like in regular CHR, if one of H_1 (in case of a *simplification rule*) or H_2 (*propagation rule*) is empty, we omit the " \ ". For propagation rules we use "$\Longrightarrow$" instead of "$\Longleftrightarrow$". At least one of the positive heads has to be non-empty. The *guards* (the *positive guard* G and the negated guards G_i) are conjunctions of built-in constraints. Empty guards (*true*) may be omitted together with the " | " symbol. Finally, the *body* (B) is a conjunction of built-in and CHR constraints.

2.2 Negated heads as extra precondition

We now present the preconditions for applying a CHR$^{\neg}$ rule. When and whether an applicable rule will actually be applied is determined by the operational semantics (covered in Sections 2.3 and 3).

A CHR$^{\neg}$ rule without negated heads is applicable if it would be in regular CHR. A negated head adds an extra precondition: a rule may *not* be applied if the constraint store contains CHR constraints that match the negated head. More precisely: a rule of the form (1) is applicable if

$$\exists \bar{p}\; \exists S_r \Big(S = H_1 \uplus H_2 \uplus S_r \;\;\wedge\;\; G \;\;\wedge\;\; \textstyle\bigwedge_{i=1}^{k} \neg\exists \bar{n}_i \big(N_i \mathrel{\underline{\Subset}} S_r \;\wedge\; G_i \big) \Big)$$

where S is a multiset representing the constraint store, $\bar{p}$ are the variables occurring in H_1, H_2, or G, and $\bar{n}_i$ are the variables occurring in N_i or G_i but not in $\bar{p}$ (the *positive* variables), and $\uplus$ and $\mathrel{\underline{\Subset}}$ denote multiset union and subset.

Example 1. The following $\mathrm{CHR}^{\neg}$ rule expresses that "*If* a person `X` is not married, *then* that person is single": `person(X) \\ married(X) ==> single(X).`

Variables bound by the positive head or guard (e.g. `X` in the above example) can be used in a negated head. Using variables introduced in the positive *guard* (this arguably is rarely needed) breaks the left-to-right reading of a $\mathrm{CHR}^{\neg}$ rule. By contrast, a negated head *cannot* bind variables to be used in the right-hand side of a rule. The intuition is that a negated head describes something that *is not there*. Therefore the scope of a variable defined in a negated head is limited to the head itself (and its guard):

Example 2 (Variable scope). `find_singles \\ married(X) ==> single(X).`

The rule in this example is applicable if `find_singles` is in the store and there is *no* `married/1` constraint in the store *at all*. When applied, it adds a `single(X)` constraint, where `X` is a fresh variable. This is probably not the intended meaning.

Example 3. Negated heads allow more concise and declarative programs. Consider the following rules from an implementation of Dijkstra's algorithm [18]:

```
dist(N,L), edge(N,N2,W) ==> L2 is L+W, relabel(N2,L2).
dist(N,_) \ relabel(N,_) <=> true.
relabel(N,L) <=> doi(N,L).
```

These rules can be rewritten to just one rule, eliminating both the auxiliary constraint `relabel/2` and the dependency on the execution order of ω_r:

```
dist(N,L), edge(N,N2,W) \\ dist(N2,_) ==> L2 is L+W, doi(N2,L2).
```

Example 4 (Distinct constraints). Consider the following $\mathrm{CHR}^{\neg}$ program:

```
parent(X,Y) \ parent(X,Y) <=> true.
parent(X,Y) \\ parent(X,_) ==> only_child(Y).
```

The first rule states that we never keep more than one constraint to describe that "`X` is a parent of `Y`". We say the `parent` constraint has *set semantics* (in contrast to the default *multiset* semantics). For the simpagation rule to be applicable, CHR requires the two constraint instances matching the positive head to be different (with identical arguments). For negated heads, we opted for something similar: constraints already used in the matching of the positive head are not allowed in the matching of the negated head. Hence, the second rule expresses "*If* `Y` is a child of `X` and there is no *other* child of `X`, *then* `Y` is an only child". We refer to this as the *distinct constraints* matching strategy (see [20]).

Example 5. The rule `p \\ p ==> q` should not be read as "*If* `p` and ¬`p`, *then* `q`" (a rule for which the antecedent never holds), since we are introducing negation *as absence* and not the classical logical negation. It should also not be read as "*If* `p` is present and `p` is absent, *then* add `q`" (a rule for which the antecedent also never holds), because of the *distinct constraints* matching strategy. A correct reading is: "*If* `p` is present, and there is no *other* `p` in the store, *then* add `q`", or, in other words: "*If* there is *exactly one* `p` constraint, *then* add `q`".

Negated guards. Up till now we have only considered negated heads without guards — although non-singleton variables in (negated) heads are implicit guards that can be made explicit: for example, the rule in Example 1 is short for:

```
person(X) \\ married(X0) | X0 == X  ==> single(X).
```

Example 6. In general, a negated head can have an arbitrary guard. The following CHR$^{\neg}$ rule defines the query-constraint `get_min/1`:

```
c(X) \ get_min(Min) \\ c(Y) | Y < X <=> Min = X.
```

This rule reads: "*If* collection `c` contains an element `X` and there is no (other) element `Y` in `c` with `Y < X`, *then* `X` is the minimum". Without negation:

```
c(X) \ get_min(Min) <=> current(X,Min).
c(X) \ current(Current,Min) <=> X < Current | current(X,Min).
current(Current,Min) <=> Min = Current.
```

Negated conjunctions. Like a positive head, a negated head is a *conjunction.*

Example 7. A half-sibling is a sibling by one parent but not by both:

```
parent(P1,X), parent(P1,Y) \\ parent(P2,X), parent(P2,Y)
 ==> half_sibling(X, Y).
```

Multiple negated heads. So far we have only considered CHR$^{\neg}$ rules with a single negated head. In general, any number of negated heads can be used.

Example 8. If a parent of `X` is married to a parent of `Y`, but `X` and `Y` do not share a (biological) parent, then `X` and `Y` are step-siblings:

```
parent(P1,X), parent(P2,Y), married(P1,P2)
   \\ parent(P2,X)   \\ parent(P1,Y)     ==> step_sibling(X,Y).
```

Example 9 (Variable scope 2). The new variables in a negated head are local:

```
c(L), c(U) \ get_bounds(LB,UB)
   \\ c(Z) | Z < L   \\ c(Z) | Z > U     <=> LB = L, UB = U.
```

The last line in the above rule is equivalent to:

```
   \\ c(Z1) | Z1 < L   \\ c(Z2) | Z2 > U   <=> LB = L, UB = U.
```

2.3 Triggering on negated heads

We know now when CHR$^{\neg}$ rules with negated heads are *applicable*. In this section we aim to give some intuition about when they should be *applied* (in anticipation of Section 3, where we formally define the operational semantics of CHR$^{\neg}$).

Essentially, the operational semantics of CHR [6] determines that a rule can fire whenever a CHR constraint is added that occurs in its (positive) head, or when a change in the built-in constraint store causes its (positive) guard to succeed. In the latter case we say that the guard *triggers* the rule. We could simply keep this semantics and treat negated heads as an extra, *passive* condition of rule applicability, much like a positive guard that does not trigger. But then many applicable CHR$^{\neg}$ rules would never be applied.

We allow a rule to fire, not only when its positive head and guard become satisfied, but also when its guarded negated head(s) become satisfied. Symmetric to the positive case, we distinguish two causes for a guarded negated head to become satisfied. The first, and most obvious, is the removal of one of the negatively occurring constraints. The second is related to negated guards. We say that the removal resp. the negated guard *triggers* the rule.

Triggering on removal. Figure 1 lists a CHR$^{\neg}$ program to maintain reachability information in a directed graph. The constraint store contains `node/1` and `edge/2` constraints, representing a dynamic graph. It is useful to imagine the listed program as part of a larger program that inserts and removes `node/1` and `edge/2` constraints at arbitrary points throughout the program.

The `set_sem_*` rules enforce set semantics. Only the `edge/2` constraint has a multiset semantics, implying that parallel edges are allowed. We use an auxiliary constraint `path(From,Over,To)` to represent that a path exists between node `From` and node `To` that starts with `edge(From,Over)`. The rules `trivial_path` and `extend_path` generate `path(A,_,B)` constraints for every path between `A` and `B`. The complementary rules `broken_tr` and `broken` ensure that, if a path is broken (because some edge or node is removed), all `path` constraints depending on the broken path are removed recursively. Finally, the last three rules use a

reachability.chr

```
set_sem_nodes @ node(A) \ node(A) <=> true.
missing_from  @ edge(A,_) \\ node(A) <=> true.
missing_to    @ edge(_,B) \\ node(B) <=> true.

set_sem_paths @ path(A,B,C) \ path(A,B,C) <=> true.
trivial_path @ node(A) ==> path(A,A,A).
extend_path  @ edge(A,B), path(B,_,C) ==> A \== B, A \== C | path(A,B,C).
broken_tr    @ path(A,_,_) \\ node(A) <=> true.
broken       @ path(A,B,C) \\ edge(A,B), path(B,_,C) <=> A \== B | true.

set_sem_reaches @ reaches(A,B) \ reaches(A,B) <=> true.
path            @ path(A,_,B) ==> reaches(A,B).
no_paths        @ reaches(A,B) \\ path(A,_,B) <=> true.
```

Fig. 1. CHR$^{\neg}$ program to maintain reachability in dynamic directed graphs

similar pattern to make sure that there is a `reaches(A,B)` constraint in the store if *and only if* a path exists between `A` and `B`.

This program illustrates the usefulness of triggering on removal: not only can we maintain correct reachability information if new paths are *created* (by adding edges or unifying nodes), in CHR$^{\neg}$ we can write *complementary rules* to keep this information *consistent* if edges (and thus paths) are *removed*. Another usage pattern shown is what we could call *garbage collection*. Obviously there is no need to keep an edge constraint if one of its end nodes is removed. In CHR$^{\neg}$ we can easily add rules (`missing_from` and `missing_to`) to remove such edges. This removal then recursively triggers the removal of all other redundant information (paths, reachability, . . .).

Example 10. Consider adding the following rule to the program in Figure 1:

```
strongly_connected, node(A), node(B) \\ reaches(A,B) ==> edge(A,B).
```

This rule forces the graph to be strongly connected if the `strongly_connected` constraint is in the store. It adds edges until every node reaches every other node. When edges (or nodes) are removed, the resulting removal of `reaches/2` constraints triggers the addition of edges until strong connectivity is restored.

Example 11. A well-known CHR pattern to maintain the minimal element of a collection of `c/1` constraints consists of the following two rules:

```
c(X) ==> min(X).
min(X) \ min(Y) <=> X ≤ Y | true.
```

Unfortunately these rules do not consistently keep the minimum if elements are removed. In CHR$^{\neg}$ we can generalize the above pattern to deal with removal:

```
min(X) \\ c(X) <=> true.
c(X) \\ c(Y) | Y < X ==> min(X).
min(X) \ min(Y) <=> X ≤ Y | true.
```

When the minimal element `c(X)` is removed, the corresponding `min(X)` constraint is removed and replaced by a new minimum.

Triggering on negated guards. A guarded negated head can also become satisfied by a (non-monotonic) change in the built-in store. The addition of a built-in constraint may cause the guard of a negated head to become false for all negated head matchings.

Example 12. Consider this slightly altered version of the last rule of Example 11

```
c(X) \\ c(Y) | Y @< X ==> min(X).
```

where `'@<'` is the (Prolog) built-in for *less than in the standard order of terms*. Consider the query "`c(A), c(B), A=g`". Adding `c(A)` propagates `min(A)`. Say `A @< B`, so adding `c(B)` does not affect the minimum. Then, unifying `A` with `g` grounds the first constraint. This should trigger the above rule, since now the negated head is satisfied for `c(B)` — not because `c(g)` is removed, but because `g @< B` no longer holds (while `A @< B` did).

This type of triggering can only happen if guards behave non-monotonically, i.e. are true and become false. In host-languages like Prolog, such guards are rather exceptional (which is why the above example is a little far-fetched). Other examples of non-monotonic guards in Prolog are "`var(X)`" which becomes false when `X` is instantiated, "`X \== Y`" which becomes false when `X` and `Y` are unified, and dynamic predicate calls in programs that use `retract/1`. In other host languages (e.g. Java [21]), this type of guards might occur more frequently.

3 Formal Semantics

In this section we formalize the semantics of a $\mathrm{CHR}^{\neg}$ program by defining a refined operational semantics. This semantics reflects several decisions already motivated in the previous section, like rule applicability and triggering of negated heads. Other aspects covered in this section are execution order and the role of the propagation history. These aspects are illustrated in an example derivation.

The formulation of the semantics is based on [6], where Duck et al. presented a similar refined semantics ω_r for regular CHR. Much like [6], we also defined a theoretical operational semantics for $\mathrm{CHR}^{\neg}$, and proved that the refined semantics presented in this section is an instance thereof. These results can be found in the extended version of this paper [20].

3.1 $\omega_r^{\neg}$: the refined operational semantics of $\mathrm{CHR}^{\neg}$

The $\omega_r^{\neg}$ semantics is formulated as a state transition system. Transition rules define the relation between subsequent execution states in a $\mathrm{CHR}^{\neg}$ derivation. We use $+\!\!+$ for sequence *concatenation* and $\sqcup$ for *disjoint union* of sets.

Execution state. Formally, an execution state of $\omega_r^{\neg}$ is a tuple $\sigma = \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$. All parts besides the *execution stack* $\mathbb{A}$ are defined as in [6]. The CHR constraint *store* $\mathbb{S}$ is a set of *identified* CHR constraints that can be matched with the rules. An *identified* CHR constraint $c\#i$ is a CHR constraint c associated with a unique *constraint identifier* i which is used to distinguish otherwise identical constraints. The counter n represents the next free integer that can be used to identify a CHR constraint. We introduce the functions $chr(c\#i) = c$ and $id(c\#i) = i$, and extend them to sequences and sets in the obvious manner. The identifiers are used by the *propagation history* $\mathbb{T}$, a set of tuples, each recording the identities of the CHR constraints that fired a rule, and the name of the rule itself. The primary function of this history is to prevent trivial non-termination for propagation rules; its role in $\omega_r^{\neg}$ will be further analyzed later in this section. Finally, the *built-in constraint store* $\mathbb{B}$ is an abstract logical conjunction of constraints, modeling all built-in constraints that have been passed to the underlying solver.

The execution stack $\mathbb{A}$ is used to treat constraints as procedure calls — as in ω_r. The top-most element is called the *active constraint.* Each newly added CHR constraint initiates a search for partner constraints to match the heads of the rules. Adding a built-in constraint initiates a similar search for applicable rules. As with a procedure, when a rule fires, other constraints (its body) are executed, and execution does not return to the current active constraint *until* these calls have finished. This approach is used because it corresponds closely to that of the stack-based programming languages to which CHR is compiled. Like ω_r, the $\omega_r^{\neg}$ semantics fixes the order in which searches for matching rules are conducted. That is why constraints on the execution stack can become *occurrenced*. The *positive* occurrences of constraints in the heads of the rules are numbered in a top-down, right-to-left manner . An *occurrenced* identified CHR constraint $c\#i:j$ indicates that only matches with positive occurrence j of constraint c should be considered when the constraint is active.

Thus far, the execution state corresponds to that of ω_r. However, in $\mathrm{CHR}^{\neg}$, rules also trigger on constraint *removal*. This translates to a slight modification in the definition of the execution stack: formally, $\mathbb{A}$ is a sequence of constraints, (occurrenced) identified CHR constraints *and (occurrenced) negated CHR constraints*. *Negated CHR constraints* are denoted as $\neg c$, where c is a CHR constraint (this notation is extended to sequences in the usual way). An *occurrenced* negated CHR constraint $\neg c : j$ denotes that *the j-th rule where c occurs negatively* is considered.

Example 13. Recall from Example 11 the $\mathrm{CHR}^{\neg}$ pattern to maintain the minimum of a collection. These three $\mathrm{CHR}^{\neg}$ rules rely on rule order to renew the minimum after the current minimal element is removed: *first* the old minimum is removed by the first rule, *then* the second rule adds the new minimum. Both rules trigger on the same removal. If the second rule would be applied first, we could get undesired behavior: two `min/1` constraints would be in the store, which allows application of the third rule, removing the *new* minimum!

Transition rules. Given an initial goal sequence G, the *initial execution state* σ_0 is $\langle G, \emptyset, true, \emptyset\rangle_1$. Execution proceeds by exhaustively applying transitions to σ_0, until the built-in store is unsatisfiable or no more transitions are applicable.

The transition rules of $\omega_r^{\neg}$ are listed in Figure 2. Besides the relatively straightforward extensions to deal with negation, the most important addition to ω_r is the **AllowReapply** transition. We say a (propagation!) rule *reapplies* if it applies with a combination of positive constraints that has already caused the rule to fire earlier in the derivation. In ω_r, rules never reapply because of the propagation history. In $\omega_r^{\neg}$ however, a rule can become reapplicable because of the **AllowReapply** transition. This new transition removes a propagation history tuple as soon as the rule, whose application introduced the tuple, is no longer applicable. Because the rule has to be inapplicable, trivial nontermination caused by infinite propagation is still avoided — the original motivation for the history.

There are two ways to get a situation in which **AllowReapply** is applicable: 1) one of the constraints used in the matching of the positive head has been removed from the constraint store; or 2) the applicability condition no longer holds for the particular rule and combination of constraints given in the tuple. Clearly the first case is a kind of garbage collection and does not affect derivations (cf.

0. AllowReapply $\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T}'\rangle_n$ where $\mathbb{T}' = \mathbb{T} \setminus \{h\}$ and $\forall \mathbb{A}'' : \langle \mathbb{A}'', \mathbb{S}, \mathbb{B}, \mathbb{T}'\rangle_n \not\rightarrowtail \langle \mathbb{A}', \mathbb{S}', \mathbb{B}', \mathbb{T}\rangle_n$. No other transition is applicable if this one is.

1. Solve $\langle [b|\mathbb{A}], S_0 \sqcup S_1, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail \langle \{[x, \neg x] | x \in S_1\} +\!\!+ \mathbb{A}, S_0 \sqcup S_1, b \wedge \mathbb{B}, \mathbb{T}\rangle_n$ where b is a built-in constraint and $vars(S_0) \subseteq fixed(\mathbb{B})$, the variables fixed by $\mathbb{B}$.

2. Activate Positive $\langle [c|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail \langle [c\#n\!:\!1|\mathbb{A}], \{c\#n\} \sqcup \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_{n+1}$ where c is a CHR constraint.

3. Reactivate Positive $\langle [c\#i|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail \langle [c\#i\!:\!1|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ where c is a CHR constraint (re-added to A by **Solve** but not yet active).

4. Activate Negated $\langle [\neg c|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail \langle [\neg c\!:\!1|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_{n+1}$

5. Drop $\langle [x : j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail \langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ where x is a (positive) identified CHR constraint $c\#i$ and there is no j-th positive occurrence of c, or x is a negated constraint $\neg c$ and there are less than j rules where c occurs negatively.

6. Simplify $\langle [c\#i\!:\!j|\mathbb{A}], \{c\#i\} \sqcup H_1 \sqcup H_2 \sqcup H_3 \sqcup S, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail$
$\langle \neg(H_2' +\!\!+ [d] +\!\!+ H_3') +\!\!+ B +\!\!+ \mathbb{A}, H_1 \sqcup S, \theta \wedge \mathbb{B}, \mathbb{T}'\rangle_n$ where the j-th (positive) occurrence of c is d in a (renamed apart) rule r of the form

$$r \ @\ H_1' \setminus H_2', d, H_3' \ \backslash\backslash\ N_1' \mid G_1 \ \backslash\backslash\ \ldots \ \backslash\backslash\ N_k' \mid G_k \Longleftrightarrow G \mid B$$

and there exists a matching substitution θ such that $c = \theta(d)$, $chr(H_i) = \theta(H_i')$, and $applicable(G, \bar{N}', \bar{G}, \mathbb{B}, \theta, S)$ holds.
Let $t = (id(H_1) +\!\!+ id(H_2) +\!\!+ [i] +\!\!+ id(H_3), r)$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.

7. Propagate $\langle [c\#i\!:\!j|\mathbb{A}], \{c\#i\} \sqcup H_1 \sqcup H_2 \sqcup H_3 \sqcup S, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail$
$\langle \neg(H_3') +\!\!+ B +\!\!+ [c\#i\!:\!j|\mathbb{A}], \{c\#i\} \sqcup H_1 \sqcup H_2 \sqcup S, \theta \wedge \mathbb{B}, \mathbb{T}'\rangle_n$ where the j-th (positive) occurrence of c is d in a (renamed apart) rule r of the form

$$r \ @\ H_1', d, H_2' \setminus H_3' \ \backslash\backslash\ N_1' \mid G_1 \ \backslash\backslash\ \ldots \ \backslash\backslash\ N_k' \mid G_k \Longleftrightarrow G \mid B$$

and there exists a matching substitution θ such that $c = \theta(d)$, $chr(H_i) = \theta(H_i')$, and $applicable(G, \bar{N}', \bar{G}, \mathbb{B}, \theta, S)$ holds.
Let $t = (id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3), r)$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.

8. TriggerNegated $\langle [\neg c\!:\!j|\mathbb{A}], H_1 \sqcup H_2 \sqcup S, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail$
$\langle \neg(H_2') +\!\!+ B +\!\!+ [\neg c\!:\!j|\mathbb{A}], H_1 \sqcup S, \theta \wedge \mathbb{B}, \mathbb{T}'\rangle_n$ where

$$r \ @\ H_1' \setminus H_2' \ \backslash\backslash\ N_1' \mid G_1 \ \backslash\backslash\ \ldots \ \backslash\backslash\ N_k' \mid G_k \Longleftrightarrow G \mid B$$

is a renamed apart instance of the j-th rule where c occurs negatively, and there exists a matching substitution θ such that $chr(H_i) = \theta(H_i')$, and $applicable(G, \bar{N}', \bar{G}, \mathbb{B}, \theta, S)$ holds. Let $t = (id(H_1) +\!\!+ id(H_2), r)$, then $t \notin \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \cup \{t\}$.

9. Default $\langle [x\!:\!j|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n \rightarrowtail \langle [x\!:\!j+1|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T}\rangle_n$ if the current state cannot fire any other transition (x can be either positive ($c\#i$) or negated ($\neg c$)).

The *applicability condition* is defined as follows: $applicable(G, \bar{N}, \bar{G}, \mathbb{B}, \theta, S) =$

$$\mathcal{D}_b \models \mathbb{B} \rightarrow \bar{\exists}_{\mathbb{B}}\Big(\theta(G) \wedge \textstyle\bigwedge_{i=1}^{k}\Big(\forall X \subseteq S : \neg\exists\eta : chr(X) = (\eta\theta)(N_i) \wedge (\eta\theta)(G_i)\Big)\Big)$$

where $\mathcal{D}_b$ is the built-in constraint domain and η are matching substitutions.

Fig. 2. The transition rules of the refined operational semantics $\omega_r^{\neg}$ for $\mathrm{CHR}^{\neg}$.

CleanHistory in [19]). However, in the second case, removing history tuples may allow reapplication of a rule. Consider a rule of the form $P \setminus\setminus N \Longrightarrow B$. Suppose this rule is applied at some point in the derivation. Later on, the constraint N is added, so the rule becomes inapplicable and the **AllowReapply** transition is applied. Now, if N is removed, the rule can be applied *again* (provided, of course, that P is still in the store). Another example of reapplication is given in Section 3.2.

There are also less obvious ways for a rule to become reapplicable: 1) a non-monotonic negated guard: a guard which is first false, allowing rule application, then becomes true, resulting in an **AllowReapply**, and then becomes false again, causing rule reapplication; or 2) a non-monotonic positive guard — a guard which is true, then false, then true again. The second case is why $\omega_r^\neg$ is *not* a generalization of ω_r (for programs with non-monotonic guards): the **AllowReapply** transition could allow the reapplication of propagation rules *without negated heads*. However, the semantics without **AllowReapply** *is* a generalization of ω_r. We call this the *fire-once* refined operational semantics $\omega_r^{\neg 1}$.

Determinism. Like ω_r, the $\omega_r^\neg$ semantics only has two sources of non-determinism: the order in which the constraints are added to $\mathbb{A}$ in the **Solve** transition and the choice of partner constraints in **Propagate**, **Simplify** and **TriggerNegated**. The **AllowReapply** transition does not introduce additional non-determinism, because we require it to fire *immediately* when it becomes applicable.

3.2 Example derivation

Consider once more the $\text{CHR}^\neg$ program of Example 11. Suppose we name these rules $\mathtt{r}_1$ to $\mathtt{r}_3$ and add the following rule to remove an element:

```
remove @ rm(X), c(X) <=> true.
```

Figure 3 shows the derivation for the query `c(9), c(5), rm(5)`. The derivation starts by activating `c(9)`, which fires the second rule $\mathtt{r}_2$. This adds `min(9)`, which does not fire any rule. Note that firing $\mathtt{r}_2$ has added a tuple to the propagation history. After `c(9)` is dropped, the next constraint `c(5)` gets activated. Because this inserts `c(5)` into the store, rule $\mathtt{r}_2$ is no longer applicable for `c(9)`, so **AllowReapply** removes the corresponding history tuple. Next, $\mathtt{r}_2$ fires once more and adds `min(5)`. This causes r_3 to remove the old minimum, `min(9)`#2, from the store, also adding ¬`min` on top of the stack (which does not trigger any rule since `min/1` does not occur negatively). Because negated constraints are not matched, the constraint symbol alone suffices. An **AllowReapply** transition removes the newly added history tuple — this always happens after application of non-propagation rules since they immediately become inapplicable (because they remove (some of) their positive heads).

We now jump to the first time constraint removal triggers a rule. After `c(5)` is removed, `min(5)` is removed by rule r_1. This removal itself triggers nothing, but the removed `c(5)` constraint also triggers the next rule, r_2, which adds the new

$$
\begin{array}{lll}
\multicolumn{3}{l}{\langle[\mathtt{c(9)},\mathtt{c(5)},\mathtt{rm(5)}]\ ,\ \emptyset\ ,\ \emptyset\rangle_1} \\
\quad\rightarrowtail_{act} & \langle[\mathtt{c(9)}\#1\!:\!1,\mathtt{c(5)},\mathtt{rm(5)}]\ ,\ \{\mathtt{c(9)}\#1\}\ ,\ \emptyset\rangle_2 & \\
\quad\rightarrowtail_{prop} & \langle[\mathtt{min(9)},\mathtt{c(9)}\#1\!:\!1,\ldots]\ ,\ \{\mathtt{c(9)}\#1\}\ ,\ \{([1],\mathtt{r}_2)\}\rangle_2 & \\
\quad\rightarrowtail_{act} & \langle[\mathtt{min(9)}\#2\!:\!1,\ldots]\ ,\ \{\mathtt{c(9)}\#1,\mathtt{min(9)}\#2\}\ ,\ \{([1],\mathtt{r}_2)\}\rangle_3 & \\
\quad\rightarrowtail^{*} & \langle[\mathtt{c(9)}\#1\!:\!1,\mathtt{c(5)},\mathtt{rm(5)}]\ ,\ \{\mathtt{c(9)}\#1,\mathtt{min(9)}\#2\}\ ,\ \{([1],\mathtt{r}_2)\}\rangle_3 & \\
\quad\rightarrowtail^{*} & \langle[\mathtt{c(5)}\#3\!:\!1,\mathtt{rm(5)}]\ ,\ \{\mathtt{c(9)}\#1,\mathtt{min(9)}\#2,\mathtt{c(5)}\#3\}\ ,\ \{([1],\mathtt{r}_2)\}\rangle_4 & \\
\quad\rightarrowtail_{ara} & \langle[\mathtt{c(5)}\#3\!:\!1,\mathtt{rm(5)}]\ ,\ \{\mathtt{c(9)}\#1,\mathtt{min(9)}\#2,\mathtt{c(5)}\#3\}\ ,\ \emptyset\rangle_4 & \hookleftarrow \\
\quad\rightarrowtail_{prop} & \langle[\mathtt{min(5)},\mathtt{c(5)}\#3\!:\!1,\mathtt{rm(5)}]\ ,\ \{\mathtt{c(9)}\#1,\mathtt{min(9)}\#2,\mathtt{c(5)}\#3\}\ ,\ \{([3],\mathtt{r}_2)\}\rangle_4 & \\
\quad\rightarrowtail^{*} & \langle[\mathtt{min(5)}\#4\!:\!3,\ldots]\ ,\ \{\mathtt{c(9)}\#1,\mathtt{min(9)}\#2,\mathtt{c(5)}\#3,\mathtt{min(5)}\#4\}\ ,\ \{([3],\mathtt{r}_2)\}\rangle_5 & \\
\quad\rightarrowtail_{prop} & \langle[\neg\mathtt{min},\mathtt{min(5)}\#4\!:\!3,\ldots]\ ,\ \{\mathtt{c(9)}\#1,\mathtt{c(5)}\#3,\mathtt{min(5)}\#4\}\ ,\ \{([3],\mathtt{r}_2),([4,2],\mathtt{r}_3)\}\rangle_5 & \\
\quad\rightarrowtail_{ara} & \langle[\neg\mathtt{min},\mathtt{min(5)}\#4\!:\!3,\ldots]\ ,\ \{\mathtt{c(9)}\#1,\mathtt{c(5)}\#3,\mathtt{min(5)}\#4\}\ ,\ \{([3],\mathtt{r}_2)\}\rangle_5 & \\
\multicolumn{3}{l}{\quad\text{(we now jump to a state little after } \mathtt{c(5)} \text{ is removed by } \mathtt{rm(5)}\text{)}} \\
\quad\rightarrowtail^{*} & \langle[\neg\mathtt{c}\!:\!1,\mathtt{rm(5)}\#5\!:\!1]\ ,\ \{\mathtt{c(9)}\#1,\mathtt{min(5)}\#4\}\ ,\ \emptyset\rangle_6 & \\
\quad\rightarrowtail_{trneg} & \langle[\neg\mathtt{min},\neg\mathtt{c}\!:\!1,\ldots]\ ,\ \{\mathtt{c(9)}\#1\}\ ,\ \emptyset\rangle_6 & \\
\quad\rightarrowtail^{*} & \langle[\neg\mathtt{c}\!:\!2,\ldots]\ ,\ \{\mathtt{c(9)}\#1\}\ ,\ \emptyset\rangle_6 & \\
\quad\rightarrowtail_{trneg} & \langle[\mathtt{min(9)},\neg\mathtt{c}\!:\!2,\ldots]\ ,\ \{\mathtt{c(9)}\#1\}\ ,\ \{([1],\mathtt{r}_2)\}\rangle_6 & \hookleftarrow \\
\quad\rightarrowtail^{*} & \langle[]\ ,\ \{\mathtt{c(9)}\#1,\mathtt{min(9)}\#6\}\ ,\ \{([1],\mathtt{r}_2)\}\rangle_7 &
\end{array}
$$

Fig. 3. $\omega_r^{\neg}$ derivation for "`c(9), c(5), rm(5)`" (the built-in store $\mathbb{B}$ is not included).

minimum. Recall from Example 13 how we rely on *rule order* here. Finally, the new minimum is activated and added to store, and the derivation reaches a final state. One final remark: the reason r_2 could fire *again* with `c(9)`#1 matching its positive head, is because the **AllowReapply** rule removed the corresponding history tuple earlier in the derivation (we marked the relevant transitions with "$\hookleftarrow$"). This illustrates why the **AllowReapply** transition is necessary.

3.3 Implementation

We have designed and implemented a source-to-source transformation scheme from CHR$^{\neg}$ to regular CHR. In general, a CHR$^{\neg}$ program with r rules and c constraints requires $\mathcal{O}(rc)$ CHR rules using this scheme. The full transformation is described in detail in [20]. This prototype implementation has enabled us to experiment with CHR$^{\neg}$ and to gain insight in its subtleties. In particular, all examples of Section 2 are executable in our prototype implementation of CHR$^{\neg}$.

4 Discussion

In this section we discuss some theoretical and practical disadvantages and limitations of the $\omega_r^{\neg}$ semantics. We also propose approaches to overcome them.

Lost logical reading. Besides the *operational* semantics ω_t [11] and ω_r [6], CHR without negation features a *declarative* semantics: CHR rules have a *logical reading*, i.e. they correspond (in a straightforward way) to formulae in either classical first-order predicate logic [11], or linear logic [4]. This useful property is lost for $\mathrm{CHR}^{\neg}$ since negation as absence is an inherently operational concept: it seems to be impossible to integrate it in any (monotonic) logic in a natural way (much like negation as failure in logic programming languages).

Lost monotonicity. Another fundamental property of the CHR language is its monotonicity [2]: rule application is independent of context, so adding constraints cannot inhibit the applicability of a rule. The monotonicity property can be stated as follows: "if $A \rightsquigarrow B$, then $A \wedge C \rightsquigarrow B \wedge C$ for any C". Clearly, this property is lost in $\mathrm{CHR}^{\neg}$ — and with it all results that rely on it, most notably the results on confluence [2] and the notions of parallelism introduced in [12].

Unexpected behavior. Programming in $\mathrm{CHR}^{\neg}$ turns out to be challenging: the $\omega_r^{\neg}$ semantics often results in unexpected and somewhat counterintuitive program behavior. Some examples:

Example 14 (Destructive update). Let `account(X,B)` model some novel type of bank account of a `client(X)` with balance `B`. Suppose the bank wants to send an information brochure to all (new) clients that do not yet have this type of account. They add the following $\mathrm{CHR}^{\neg}$ rule:

```
send @ client(X) \\ account(X,_) ==> send_brochure_to(X).
```

Suppose the $\mathrm{CHR}^{\neg}$ program used by the bank also contains a rule of the form "`deposit(X,Amount), account(X,B) <=> account(X,B+Amount)`". This rule is an example of a common CHR pattern, often referred to as *(destructive) update*. The problem is that removing a constraint (`account(X,B)`) can trigger rules (`send`), even though the constraint will immediately be replaced with an updated version. This means that each time a client deposits an amount on his account, he receives a brochure. A brochure about an account type he already has. Some might argue this is not what the bank intended.

Example 15 (Order). Negatively occurring constraints have to be added in the right order. The following program causes each child to be declared orphan at birth because of an incorrect order in the body of the first rule:

```
birth(C,F,M) <=> child(C), father(F,C), mother(M,C).
child(C) \\ father(_,C) \\ mother(_,C) ==> orphan(C).
```

Rearranging the body constraints solves this problem. Another example is the graph program (Figure 1), where nodes have to be created before edges. In some programs there is no correct order to add constraints one at a time: in Example 4, there is no way to insert constraints `parent(P,`C_1`)` and `parent(P,`C_2`)` without propagating `only_child(`C_1`)` or `only_child(`C_2`)`. We expect that this can often be corrected: e.g. in Example 4, we could add the rule:

```
parent(P,C), parent(P,_) \ only_child(C) <=> true.
```

Example 16 (Atomicity). Suppose we want to improve the graph program of Figure 1 by adding (in front of the program):

```
add_nodes @ edge(A,B) ==> node(A), node(B).
```

The idea is to allow the creation of edges without having to add node constraints first (cf. example 15). Unfortunately, this innocent-looking rule causes unexpected behavior. Consider the query "`edge(a,b), edge(b,c)`". The first constraint propagates `node(a)` and `node(b)` because of `add_nodes`. After activating the second constraint a duplicate constraint `node(b)` will be activated, *before* `node(c)` is added to the store. The former constraint is redundant so it is removed by `set_sem_nodes`. This removal however triggers `missing_to`, which deletes `edge(b,c)`. Remember, there is no `node(c)` yet! So the final store contains `edge(a,b),node(a),node(b),node(c)` but not `edge(b,c)`. One way to solve this problem is to replace `add_nodes` with:

```
edge(A, _) \\ node(A) ==> node(A) pragma passive(node/1).
edge(_, B) \\ node(B) ==> node(B) pragma passive(node/1).
```

We have extended the `passive` pragma [13] to negated heads: rules do not trigger on the removal of negated heads that are declared `passive` (see [20]). Here the pragmas avoid making node removal impossible for non-isolated nodes.

The above example shows two issues:

– There is a disparity between the two phases of a rule application: constraints are removed from the store *atomically* (removed heads are removed together) while they are added *incrementally* (one constraint at a time).
– Removing a constraint (e.g. `node(b)`) can trigger negated heads that test on the absence of other constraints (`node(c)`).

In fact, most issues raised by the above examples are symptoms of the same, fundamental problem of integrating negation as absence in CHR while preserving the conventional ω_r semantics. In ω_r, constraints (from a query or a rule body) are added to their respective stores (user-defined or built-in) one at a time, treating each addition as a procedure call (cf. Section 3). The problem is that each of these constraint calls will look for applicable rules disregarding the constraints that might be added shortly. Each applicable rule found is fired, and each of the constraints in their body will *in turn* have no knowledge of any unactivated constraints lower on the execution stack. This is a known issue with the common operational semantics of regular CHR, but it hardly ever poses a problem in practice. However, as shown by the above examples, the integration of negation severely exacerbates the problem.

Potential solutions. Our preliminary attempts to formulate a clean logical reading for $\mathrm{CHR}^{\neg}$ have failed in both classical and linear logic. However, it may be possible to obtain a declarative semantics in a non-monotonic logic [3] like *Transaction Logic* ($\mathcal{TR}$) [14].

Additional pragmas could be added to avoid some of the practical problems shown above, but this approach is ad-hoc and seems contrary to the declarative nature of CHR. A more satisfactory solution would be based on a significantly different operational semantics for CHR (and $\mathrm{CHR}^{\neg}$) — e.g. a more breadth-first batch-like semantics or one in which the rule order is enforced more strongly than in ω_r. There has been little interest in an alternative operational semantics for CHR in the past, but it seems to be a promising area of future research.

5 Conclusion

We introduced $\mathrm{CHR}^{\neg}$, an extension of CHR with negation as absence. $\mathrm{CHR}^{\neg}$ rules can test for constraint *absence* and can fire after constraint *removal*. We proposed an operational semantics $\omega_r^{\neg}$ for $\mathrm{CHR}^{\neg}$, and formalized it as an extension of the regular CHR semantics. We illustrated the benefits and issues of $\omega_r^{\neg}$ and some of the decisions made in designing it. We developed a prototype implementation based on a source-to-source transformation form $\mathrm{CHR}^{\neg}$ to CHR [20].

The general conclusion of this exploratory work is one of mixed feelings. On the one hand, many beautiful theoretical properties of CHR are lost in $\mathrm{CHR}^{\neg}$. The issues discussed in Section 4 also indicate that programming in $\mathrm{CHR}^{\neg}$ might require a thorough understanding of the execution mechanism. This is inconsistent with the declarative nature of CHR. On the other hand, negation increases the expressiveness and conciseness of rules, and adds a nice symmetry between the operational semantics of constraint insertion and constraint removal.

Related Work. Other, related languages offer features similar to negation as absence. The rules proposed in [22] — used to update (incomplete) knowledge bases — can contain both classical negation and negation as absence. In the Petri Net area, *inhibitor arcs*, testing for the absence of tokens, are long known [5]. All major production rule systems [8, 10, 15, 16] provide negation as absence. In fact, most [10, 15, 16] allow the logical connectives `and`, `or` and `not` — arbitrary nested — on the left-hand side of their rules. In [7], an extension of production rules with *negation as failure* is motivated and explored. A more detailed overview of related work can be found in [20]. A common thread is that rules are applied atomically [9], which relaxes the practical issues discussed in Section 4.

In the context of CHR, [1] proposed the technique of *explicit negation*, which amounts to adding, for every constraint `c/n`, an auxiliary constraint `notc/n` and a rule of the form `c(`$\bar{X}$`), notc(`$\bar{X}$`) ==> fail`. This allows queries and rule bodies to *tell* the *classical* negation of a constraint. To the best of our knowledge, we are the first to examine negation *as absence* in the context of CHR.

Future Work. We expect $\mathrm{CHR}^{\neg}$ to have interesting but complicated theoretical and practical properties. Our work can be seen as a proposal for — and an early experiment in — intensified research on negation and CHR. Section 4 already indicated several possible research directions in the context of negation as absence. Other forms of negation can be investigated as well. In time, an optimized compilation of CHR$^\neg$ might be warranted. Other left-hand side extensions (disjunction, nested logical operators, ...) can also be explored.

Acknowledgements. We thank Thom Frühwirth and the members of his CHR research team, Hariolf Betz, Martin Käser, Marc Meister and Jairson Vitorino, for the interesting and relevant discussions during their recent visit to our department. We are also grateful to the anonymous referees for their helpful comments.

References

1. S. Abdennadher and H. Christiansen. An experimental CLP platform for integrity constraints and abduction. In *Intl. Conf. Flexible Query Answering Systems*, 2000.
2. S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal*, 4(2):133–165, 1999.
3. G. A. Antonelli. Non-monotonic logic. Stanford Encyclopedia of Philosophy, 2006. http://plato.stanford.edu/archives/spr2006/entries/logic-nonmonotonic/.
4. H. Betz and T. Frühwirth. A linear-logic semantics for CHR. In *11th Intl. Conf. Principles and Practice of Constraint Programming*, 2005.
5. S. Christensen and N. D. Hansen. Coloured Petri nets extended with place capacities, test arcs and inhibitor arcs. In *Appl. and Theory of Petri Nets*, 1993.
6. G. J. Duck, P. J. Stuckey, M. García de la Banda, and C. Holzbaur. The refined operational semantics of CHR. In *20th Intl. Conf. Logic Programming*, 2004.
7. P.M. Dung and P. Mancarella. Production systems with negation as failure. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):336–352, 2002.
8. C. L. Forgey. OPS5 User's Manual. Technical Report CMU-CS-81-135, Carnegie Mellon University, Dept. CS, 1981.
9. C. L. Forgey. RETE: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.
10. E. Friedman-Hill et al. Jess home page. http://www.jessrules.com/, June 2006.
11. T. Frühwirth. Theory and practice of CHR. *J. Logic Programming*, 37, 1998.
12. T. Frühwirth. Parallelizing union-find in CHR using confluence analysis. In *21st Intl. Conf. Logic Programming*, 2005.
13. C. Holzbaur and T. Frühwirth. CHR reference manual for SICStus Prolog. Technical Report TR-98-01, Austrian Research Institute for Artificial Intelligence, 1998.
14. M. Meister. A Transaction Logic Semantics for CHR. Seminar Day on CHR (http://www.cs.kuleuven.be/~toms/CHR/chr_wog.html), May 2006.
15. M. Proctor et al. JBoss Rules. http://www.jboss.com/products/rules, June 2006.
16. G. Riley et al. CLIPS home page. http://www.ghg.net/clips/CLIPS.html, 2006.
17. T. Schrijvers. K.U.Leuven CHR system. http://www.cs.kuleuven.be/~toms/CHR.
18. J. Sneyers et al. Dijkstra's algorithm with Fibonacci heaps: An executable description in CHR. In *20th Workshop on Logic Programming*, 2006.
19. J. Sneyers, T. Schrijvers, and B. Demoen. Memory reuse for CHR. In *22nd Intl. Conf. Logic Programming*, August 2006. To appear.
20. P. Van Weert, J. Sneyers, et al. To CHR$^\neg$ or not to CHR$^\neg$: Extending CHR with negation as absence. Technical Report CW 446, K.U.Leuven, Dept. CS, 2006.
21. P. Van Weert, T. Schrijvers, and B. Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In *2nd Workshop on CHR*, 2005.
22. Y. Zhang and N.Y. Foo. Towards generalized rule-based updates. In *15th Intl. Joint Conference on Artificial Intelligence*, volume 1, pages 82–88, August 1997.