To CHR$^{\neg}$ or not to CHR$^{\neg}$: Extending CHR with Negation as Absence

Peter Van Weert
Jon Sneyers
Tom Schrijvers
Bart Demoen

Report CW 446, May 2006



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

To CHR$^{\neg}$ or not to CHR$^{\neg}$: Extending CHR with Negation as Absence

Peter Van Weert
Jon Sneyers
Tom Schrijvers
Bart Demoen

Report CW 446, May 2006

Department of Computer Science, K.U.Leuven

Abstract

In this exploratory paper, we introduce CHR$^{\neg}$, an extension of the CHR language with negation as absence, and we illustrate how the added expressiveness allows CHR programmers to write more declarative and concise rules. We show the difficulties of integrating negation with the conventional execution mechanism of CHR. A formal operational semantics for CHR$^{\neg}$ is presented, and its theoretical and practical implications are evaluated critically. We introduce a source-to-source transformation from CHR$^{\neg}$ to regular CHR.

Keywords : Constraint Handling Rules, negation, language design, semantics, source-to-source transformation.
CR Subject Classification : D.3.2 [Programming Languages] Language Classifications — Constraint and logic languages.

Table of Contents

1 Introduction 1
2 CHR$^{\neg}$ by example: why CHR$^{\neg}$? 2
2.1 Syntax 2
2.2 Negated heads as precondition 3
2.2.1 Negated guards 4
2.2.2 Negated conjunctions 5
2.2.3 Multiple negated heads 5
2.3 Triggering on negated heads 5
2.3.1 Triggering on removal 6
2.3.2 Triggering on negative guards 8
3 Formal semantics 8
3.1 $\omega_t^{\neg}$: the abstract operational semantics of CHR$^{\neg}$ 9
3.1.1 Execution state 9
3.1.2 Transition rules 9
3.1.3 Applicability condition 9
3.1.4 Fire-once versus fire-many 10
3.2 $\omega_r^{\neg}$: the refined operational semantics of CHR$^{\neg}$ 11
3.2.1 Execution state 11
3.2.2 Transition rules 12
3.2.3 Determinism 12
3.3 Example derivation 12
4 Discussion: why not CHR$^{\neg}$? 14
4.1 Lost properties 14
4.2 Active versus passive negated heads 15
4.3 Problems of $\omega_r^{\neg}$ 16
4.3.1 Potential solutions 17
5 Transforming CHR$^{\neg}$ to CHR 18
5.1 Explicit identifiers 18
5.2 Explicit histories 18
5.2.1 The **AllowReapply** transition 19
5.3 Eliminating negated heads 21
6 Related work 22
7 Conclusion 25
7.1 Future work 25

To CHR$^{\neg}$ or not to CHR$^{\neg}$: 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 CHR$^{\neg}$, an extension of the CHR language with negation as absence, and we illustrate how the added expressiveness allows CHR programmers to write more declarative and concise rules. We show the difficulties of integrating negation with the conventional execution mechanism of CHR. A formal operational semantics for CHR$^{\neg}$ is presented, and its theoretical and practical implications are evaluated critically. We introduce a source-to-source transformation from CHR$^{\neg}$ to regular CHR. This is an extended version of [31].

1 Introduction

Constraint Handling Rules (CHR) [15] is a high-level, declarative programming language extension based on multi-headed committed-choice rules.

CHR is a constraint-driven language: *adding* constraints can cause rules to fire, depending on the *presence* of earlier constraints. The *removal* and *absence* of constraints has never received much attention in the past. This stems from the original intended purpose of CHR: the high-level declaration of application-tailored constraint solvers. In theory, CHR only removes constraints to rewrite the constraint store in an equivalent, but simpler form. Hence the term *simplification* rules. In practice however, CHR is increasingly being used as a general programming language, in a wide range of applications. Here it is often perceived that the lack of information is meaningful; i.e., one wishes to fire a rule if certain CHR constraints are *not* in the constraint store. Similarly, from time to time we want to apply a rule if constraints are *removed*.

Closely related to CHR are production rules (or business rules, as they are fashionably being called now). These rule languages traditionally offer some form of *negation* for the above purposes. The CHR programmer however is forced to introduce auxiliary constraints and/or rules to test for the absence of constraints. Such patterns emerge frequently in realistic CHR programs (e.g. [27]), and typically have to rely on rule order [11]. This leads to more verbose and less declarative programs. Reacting on the removal of constraints even requires numerous cross-cutting changes to a CHR program: you have to add auxiliary constraints to the body of every rule that removes a constraint. Clearly, these ad-hoc solutions are very cumbersome and error-prone.

Even a decade ago, CHR programmers have felt the need for negation as absence. An early illustration can be found in a CHR version of the classical *Monkey and Banana* forward chaining rules which was written by Christian Holzbaur in 1997.

* 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).

The guard of some of the CHR rules uses an auxiliary Prolog predicate `not/1` which is defined as follows:

```
not(Fact) :- find_constraint(Fact,_), !, fail.
not(_).
```

This corresponds exactly to negation as absence: `not(C)` succeeds iff the CHR constraint `C` is not in the store. Note that `find_constraint/2` is a predicate which is available in the reference CHR implementation in SICStus Prolog [5].

In this paper we introduce and explore an extension of CHR, called CHR$^\neg$, which adds the feature of negation as absence, i.e. in CHR$^\neg$, the left-hand side of rules consists of negated heads as well as the usual positive heads. This will allow the CHR programmer to write more succinct and declarative programs.

Negation as absence. We want to point out that the kind of negation we are investigating in this paper is negation *as absence*. Other kinds of negation can be considered as well, for example something closer to the classical negation to express that a constraint *does not hold*, or something like negated implication checking that expresses that a constraint *is not entailed* by the current constraint store. Although those 'other negations' are in our opinion well worth investigating, this work focusses solely on negation as absence.

Overview. We assume the reader to be familiar with the basic syntax and semantics of CHR [15]. 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. A transformation scheme from CHR$^\neg$ to CHR is introduced in detail in Section 5. Finally, Section 6 discusses some related work and Section 7 concludes.
A shorter version of this paper was submitted to the third CHR workshop [31].

2 CHR$^\neg$ by example: why CHR$^\neg$?

In this section we will introduce CHR$^\neg$ by means of several small examples. First we present its syntax as a minimal extension of the regular CHR syntax [15].

2.1 Syntax

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

$$name@\ H_1 \setminus H_2 \setminus\!\setminus N_1 \mid G_1 \setminus\!\setminus \ \ldots \ \setminus\!\setminus N_k \mid G_k \Longleftrightarrow G \mid B \qquad (1)$$

All *heads* (H_1, H_2 and the N_i's) are conjunctions of CHR constraints. New with respect to regular CHR are the *negated heads* N_i. 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. The heads inherited from CHR — H_1 (the *kept* constraints) and H_2 (the *removed* constraints) — are now referred to as *positive heads*. Depending on the context, we will also use the term *positive head* to denote both H_1 and H_2 together. 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 " $\setminus$ ". The requirement that at least one of the positive heads has to be non-empty remains. For propagation rules we also use "$\Longrightarrow$" instead of "$\Longleftrightarrow$".

All *guards*, i.e. the G_i's and the *positive guard* G, are conjunctions of built-in constraints. An empty guard, *true*, can always be omitted together with the " | " symbol. Finally, the *body* (B) is a conjunction of built-in and CHR constraints.

2.2 Negated heads as precondition

A CHR$^\neg$ rule without negated heads is applicable[1] under exactly the same conditions as a rule 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. Formally: a rule of the form (1) is applicable if

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

where S is 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).

Example 1. The following 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 (e.g. `X` in the above example) can be used in a negated head. The same applies for variables introduced in the positive guard, even though this breaks the left-to-right reading of a CHR$^\neg$ rule. Fortunately, this will only be needed in exceptional cases. By contrast, a negated head *cannot* bind variables to be used in the right-hand side of a rule. The intuition is that a negative 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. `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 (Need for auxiliary rules). Negated heads allow more concise and declarative programs. Consider the following rules, taken from a CHR implementation of Dijkstra's algorithm [27]:

```
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).
```

Using negation, these three rules can be rewritten as one rule, eliminating the need for the auxiliary constraint `relabel/2` and the dependency on the execution order of the refined semantics:

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

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

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

The first rule states that we will 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 with the default *multiset* semantics). In CHR the two positive heads are required to be different constraints (with identical arguments) for the simpagation rule to be applicable. For negated heads, we opted for something similar: constraints used in

[1] When/whether an applicable rule will actually be applied is determined by the operational semantics given to the program. This aspect will be explored by example in 2.3, and formally defined in Section 3.

the matching of the positive head are *not* allowed to falsify the negated head. This means the second rule can be read as "*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" for negated heads.

So a rule `p \\ p ==> q` should not be read as "*If* `p` and ¬`p`, *then* `q`" (a rule for which the antecedent never holds), but rather: "*If* there is *exactly one* `p` constraint in the store, *then* `q`". The alternative would be to allow negated heads to match all constraints in the store, *including* those used in the positive matching. Under this alternate strategy, the above rule would never be applicable, and its equivalent would have to be written as `p \\ p, p ==> q`. So the latter strategy results in more verbose programs. Therefore, and because the alternative is seldom needed, we choose for the *distinct constraints* matching strategy.

Furthermore, a nice *complementarity* property holds in this strategy, that does not hold for the alternative. Consider programs of the form

$$r_1 \ @ \ H_1(\bar{x}_1), H_2(\bar{x}_2) \Longrightarrow G_1(\bar{x}_1), G_2(\bar{x}_2) \mid B_1$$

$$r_2 \ @ \ H_1(\bar{x}_1) \setminus\setminus H_2(\bar{x}_2) \mid G_2(\bar{x}_2) \Longrightarrow G_1(\bar{x}_1) \mid B_2$$

where $H_i(\bar{x}_i)$ is an arbitrary conjunction of CHR constraints using the variables $\bar{x}_i$. If the CHR store contains $H_1(\bar{x}_1)$ and the guard $G_1(\bar{x}_1)$ holds, then either rule r_1 fires, or rule r_2 fires (exclusive or). Note that without negation we would have to rely on rule order to express this.

Example 5 (Complementary rules). Consider the following CHR¬ program:

```
city(X), city(Y), dist(X,Y,D) ==> D < 100 | neighbor_city(X,Y).
city(X) \\ city(Y), dist(X,Y,D) | D < 100 ==> isolated_city(X).
```

Because of the above property, this program has the meaning indicated by the predicate names: for every city `X`, `neighbor_city(X,Y)` constraints are propagated for every city within 100 kilometers, and if there are no such "close" cities, `X` is declared to be isolated. If we would have chosen the alternative matching strategy, the complementarity property would not hold: in this example, no city would be "isolated" since every city has at least one "close" neighbor: the city itself.

2.2.1 Negated guards

Up till now we have ignored guards. Most (negated) heads contain syntactic sugar for guards though. For example, the rule in Example 1 is short for:

Example 6. `person(X) \\ married(X0) | X0 == X ==> single(X).`

We say the guard `X0 == X` is *implicit* in Example 1 and *explicit* in Example 6. In general, a negative head can have an arbitrary (explicit) guard:

Example 7. The following CHR¬ rule defines the query-constraint `get_min(-)`:

```
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". Compare this single rule with the common CHR pattern to retrieve the minimal element in regular CHR:

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

Example 8 (Positive versus negative guard). Now consider the slightly modified rule, where we moved the guard on `Y` out of the negated head:

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

It is important to see why, semantically, this is *not* a correct rule. This is again due to the scoping of variables we defined earlier, but it is something well worth repeating. The rule is equivalent with:

```
c(X) \ min(Min) \\ c(Y1) | Y1 == Y <=> Y < X | Min = X.
```

with `Y` some new variable, which clearly does not have the intended meaning.

2.2.2 Negated conjunctions

Like the positive head, a negated head is in fact a conjunction of CHR constraints. Its matching is also defined analogously.

Example 9. The following rule contains four implicit guards (and one explicit):

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

Note that the explicit guard (`P1 \== P2`) is redundant if the `parent/2` constraint has set semantics like in Example 4.

2.2.3 Multiple negated heads

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

Example 10. Yet another family relation can be specified conveniently using two negated heads:

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

Example 11. Using the same variable in more than one negated head does *not* result in an implicit guard:

```
c(X), c(Y) \ min_max(Min,Max)
     \\ c(Z) | Z < X
     \\ c(Z) | Z > Y
     <=> Min = X, Max = Y.
```

This is in accordance with the scope of a new variable in a negated head we defined before: each variable with the same name outside this scope is considered a different variable.

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 common operational semantics of CHR [11] determines that a rule can fire whenever a CHR constraint occurring in its (positive) head is added,

or when a change in the built-in constraint store causes its (positive) guard to succeed. In the latter case we say that the built-in store *triggers* the guard. We could simply keep this semantics and treat negated heads as an extra, *passive* condition of rule applicability, much like a positive guard that never triggers. But then many applicable CHR$^\neg$ rules would never be applied.

Things become more interesting if we allow a rule to fire, not only when its positive head and guard become satisfied, but also when its guarded negative head(s) become satisfied. Symmetric to the positive case, we distinguish two types of causes for a guarded negative 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 built-in store *triggers* the rule (or sometimes: the negated head).

2.3.1 Triggering on removal

Figure 1 lists an example 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.

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.
```

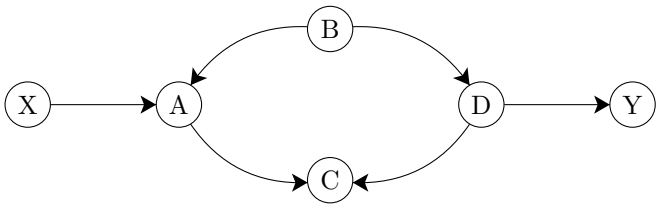
Fig. 1. CHR$^\neg$ program to maintain reachability in dynamic directed graphs

The `set_sem_*` rules enforce set semantics. Only the `edge/2` constraint has a multiset semantics. This implies 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 (after some edge or node is removed), all `path` constraints depending on the broken path are removed recursively. Finally, the last three rules use a 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 is a perfect illustration of 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*. *garbage collection* is what we could call the other usage pattern demonstrated by this ex-

ample. 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. Such removals will then in turn trigger the recursive removal of all other redundant information (paths, reachability, ...).

Example 12. To illustrate the program, consider the following graph:



The above graph corresponds to a query consisting of 6 `node/1` constraints and 6 `edge/2` constraints. After execution, `reaches(B,C)` will be in the constraint store because there is a path from `B` to `C`, e.g. `(B,A,C)`. If we now remove the edge from `B` to `A`, `reaches(B,C)` will still be in the constraint store because there still is a path: `(B,D,C)`. If we also remove the edge from `D` to `C`, `reaches(B,C)` will be removed from the constraint store. Note that at this point, `reaches(X,Y)` is not in the store. Now if we perform the unification `B=C`, one of the results will be the addition of `reaches(X,Y)` because of the new path `(X,A,C=B,D,Y)`. Finally, if we remove node `B` (which became the same node as node `C`), the only path between `X` and `Y` is broken, which causes `reaches(X,Y)` to be removed again.

Example 13. In CHR$^\neg$ the following, similar rules can be expressed:

Rule(s)	*Fires if...*	*Result*
`edge(A,_) \\ node(A) <=> true.` `edge(_,B) \\ node(B) <=> true.`	one endpoint is missing	`edge(a,a)`, `edge(a,b)`
`edge(A,B) \\ node(A) \\ node(B)` `<=> true.`	both endpoints are missing	`edge(a,a)`, `edge(a,b)`, `edge(b,c)`
`edge(A,B) \\ node(A), node(B)` `<=> true.`	*not(* two nodes exists that match the endpoints *)*	`edge(a,b)`
`edge(A,B) \\ node(A), node(B)` `<=> A \== B | true.`	the edge is not a loop and *not(* two nodes exists that match the endpoints *)*	`edge(a,a)`, `edge(a,b)`, `edge(d,d)`

To clarify the difference, the third column shows the result after the removal of `node(c)` and `node(d)` if the initial store contained the edges `edge(a,a)`, `edge(a,b)`, `edge(b,c)`, `edge(c,d)` and `edge(d,d)`, together with the four corresponding node constraints. For our graph program, the first and third variations are valid candidates. Because our nodes have set semantics, the third variant will automatically remove all loops. Loops do not influence reachability though, so this remains a perfectly acceptable alternative here. But since we regard these rules as part of a larger program, we have opted for the first alternative.

Note that in another program and/or with other constraints, one of the other alternatives might be relevant. This illustrates the rich expressiveness of CHR$^\neg$.

Example 14. 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 will add edges until every node reaches every other node. If edges (or nodes) are removed, the resulting removal of `reaches/2` constraints triggers the addition of edges until strong connectivity is restored.

The following nice example of the expressiveness of CHR$^\neg$ concludes this subsection. We will come back to it later in Example 17 of Section 3, and it will be the running example in sections 3.3 and 5.

Example 15. A well-known CHR pattern to maintain the minimal element of a collection `c` consists of the following two rules:

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

Unfortunately these rules cannot 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.
```

In this version, when the minimal element `c(X)` is removed, the corresponding `min(X)` constraint is also removed and replaced by a new minimum.

2.3.2 Triggering on negative guards

The second cause for a guarded negative head to become satisfied — the first being the removal of a constraint — is a change in the built-in store (caused by some host-language constraint) that makes a negated guard fail that used to succeed.

Example 16. Consider this slightly altered version of the last rule of Example 15

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

where `'@<'` is the (Prolog) built-in constraint defined as: $\mathtt{T}_1$ `@<` $\mathtt{T}_2$ iff term $\mathtt{T}_1$ comes before term $\mathtt{T}_2$ in the standard order of terms [1].

Suppose we have the query "`c(A), c(B), A=g`". After adding `c(A)`, the minimal (and only) element is `c(A)`. Say `A @< B`, so adding `c(B)` does not affect the minimum. Then the unification grounds the first constraint. This should trigger the above rule, since the negated head now is satisfied for `c(B)` — not because `c(A=g)` is removed, but because `g @< B` no longer holds while `A @< B` did.

Note that this type of triggering can only happen if guards behave non-monotonically, i.e. are true and become false. In logic host-languages like HAL [17] and Prolog [26], such guards are rather exceptional. This is why the above example is a little far-fetched (another example could easily be constructed using negation-as-failure). In other host languages though, like e.g. Java [30], this type of guards might occur more frequently.

3 Formal semantics

In this section we formalize the semantics of a CHR$^\neg$ program by defining both an abstract and a refined operational semantics. This semantics reflects several decisions already motivated in the previous section, like rule applicability and triggering of negative heads. Execution order and the role of propagation histories are two of

the other important facets this section focuses on. To illustrate some of the latter aspects, we conclude this section with a worked-out example derivation.

The formulation of the semantics and the notation are based upon [11]. We already now that all syntactically valid CHR programs are also syntactically valid CHR$^\neg$ programs (cf. section 2.1). In this section we also discuss the semantical relationship between CHR and CHR$^\neg$.

3.1 $\omega_t^\neg$: the abstract operational semantics of CHR$^\neg$

The $\omega_t^\neg$ semantics is formulated as a state transition system. Transition rules define the relation between an execution state and its subsequent execution state.

Sets, multisets and sequences (ordered multisets) are defined as usual. We use $+\!\!+$ for sequence *concatenation* and $\sqcup$ for *disjoint union* of sets.

3.1.1 Execution state

An *execution state* σ of $\omega_t^\neg$ is defined as a tuple $\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$. The first part, the *goal* $\mathbb{G}$, is the multiset of constraints to be rewritten to solved form. The CHR constraint *store* $\mathbb{S}$ is the 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. This number serves to differentiate among copies of the same constraint (without these identifiers, the CHR store would be a multiset). 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 *built-in constraint store* $\mathbb{B}$ is an abstract logical conjunction of constraints, modelling all built-in constraints that have been passed to the underlying solver. The *propagation history* $\mathbb{T}$ is a set of sequences, each recording the identities of the CHR constraints that fired a rule, and the name of the rule itself. Finally, the counter n represents the next free integer that can be used to identify a CHR constraint.

Given an initial goal $\mathbb{G}$, the *initial execution state* σ_0 is: $\langle \mathbb{G}, \emptyset, true, \emptyset \rangle_1$.

3.1.2 Transition rules

The theoretical operational semantics $\omega_t^\neg$ for CHR$^\neg$ programs is based on the transition rules listed in Figure 2.

A consecutive series of transition rule applications is called a *derivation*. Starting from ω_0, the transitions are applied *non-deterministically* until no more transitions are possible (a *successful* derivation), or until the built-in solver can prove $\mathcal{D}_b \models \neg\bar{\exists}_\varnothing B$ (a *failed* derivation), with $\mathcal{D}_b$ the built-in constraint domain. In both cases a *final state* has been reached.

3.1.3 Applicability condition

The first three transition rules are the same as those of ω_t, the corresponding semantics in [11]. The only exception of course is that the *applicability condition* in **Apply** takes into account negated heads:

Definition 1 (Applicability condition). *A rule with guard G and negated heads $\bar{N}$ with corresponding guards $\bar{G}$ is* applicable *under built-in constraint store $\mathbb{B}$ and a matching substitution θ, given a set of CHR constraints S (abbreviated: $applicable(G, \bar{N}, \bar{G}, \mathbb{B}, \theta, S)$), iff*

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

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

1. Solve $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{solve} \langle \mathbb{G}, \mathbb{S}, c \wedge \mathbb{B}, \mathbb{T} \rangle_n$, where c is a built-in constraint.

---

2. Introduce $\langle \{c\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{introduce} \langle \mathbb{G}, \{c\#n\} \sqcup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{(n+1)}$ where c is a CHR constraint.

---

3. Apply $\langle \mathbb{G}, H_1 \sqcup H_2 \sqcup S, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail_{apply} \langle B \uplus \mathbb{G}, H_1 \sqcup S, \theta \wedge \mathbb{B}, \mathbb{T}' \rangle_n$ where there exists a (renamed apart) rule of the form

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

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

---

4. AllowReapply $\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail \langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T}' \rangle_n$ where $h \in \mathbb{T}$ and $\mathbb{T}' = \mathbb{T} \setminus \{h\}$ and

$$\langle \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T}' \rangle_n \not\rightarrowtail_{apply} \langle \mathbb{G}', \mathbb{S}', \mathbb{B}', \mathbb{T} \rangle_n$$

Fig. 2. The transition rules of the theoretical operational semantics $\omega_t^{\neg}$ for CHR$^{\neg}$

Note that for $k = 0$ this reduces to the applicability condition of ω_t. It is therefore easily seen that the semantics that uses only these three rules is a generalization of ω_t. We call this semantics the *fire-once* theoretical operational semantics $\omega_t^{\neg 1}$ for CHR$^{\neg}$.

3.1.4 Fire-once versus fire-many

If we also add the **AllowReapply** transition, we get the $\omega_r^{\neg}$ semantics, or the *fire-many* theoretical operational semantics for CHR$^{\neg}$.

We say a (propagation!) rule *reapplies* if it is applied with a combination of positive constraints that has already caused the rule to fire earlier in the derivation. In $\omega_t^{\neg 1}$ (and ω_t) the propagation history *always* prevents this. In the presence of negation, a rule can however easily become *reapplicable* if constraints matching a negated head, that were added *after* a previous application of the rule, are removed. Therefore we argue that reapplication *should* be allowed. A clear, convincing example (based on Example 15) is given in Section 3.3. Hence, in the rest of this paper, we will consider the *fire-many* variant to be the default (theoretical) operational semantics of CHR$^{\neg}$ programs.

Reapplication is made possible by the extra transition rule, **AllowReapply**, which removes a propagation history tuple if the transition that introduced it earlier is no longer applicable. This can be because one of the constraints used in the matching has been removed from the constraint store, or because the applicability condition no longer holds for that particular rule and combination of constraints. Clearly the former case is just a form of garbage collection, and will never influence a derivation (the transition can e.g. always be applied on the history tuples of simplification and simpagation rules). Removing history tuples in the latter case however is why the **AllowReapply** transition *allows reapplication* of a rule later in the derivation.

Note that by demanding that the rule has to be inapplicable, trivial non-termination caused by infinite rule application is still avoided.

There are also less obvious causes for a rule to become reapplicable: non-monotonic negated guards and/or a non-monotonic positive guard. The latter case is why, strictly speaking, $\omega_t^{\neg}$ is *not* a generalization of ω_t (in the sense that the semantics of positive-only CHR programs is not necessarily maintained): the **AllowReapply**

transition could allow the reapplication of propagation rules *without negated heads*. The fire-once semantics $\omega_t^{\neg 1}$ (without the **AllowReapply** rule) on the other hand *is* a generalization of ω_t (and therefore probably easier to implement on top of existing systems).

Also, if we limit all guards to monotonic constraints, $\omega_t^{\neg}$ remains a generalization of ω_t. Otherwise this property is lost. Non-monotonic guards will hardly ever occur in declarative CHR host-languages like e.g. Prolog and HAL. One option would be to prohibit non-monotonic host-language statements in a guard. This however seems too restricting, certainly when CHR is embedded in less monotone languages, like e.g. Java.

3.2 $\omega_r^{\neg}$: the refined operational semantics of CHR$^{\neg}$

Now we will formulate $\omega_r^{\neg}$, a generalization of the refined operational semantics ω_r [11] for regular CHR programs. The $\omega_r^{\neg}$ semantics is also formulated as a state transition system:

3.2.1 Execution state

Formally, an execution state σ of $\omega_r^{\neg}$ is defined as a tuple $\langle \mathbb{A}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$. All parts besides the *execution stack* $\mathbb{A}$ are defined exactly as above. *The execution stack* $\mathbb{A}$ is used by $\omega_r^{\neg}$ to treat constraints as procedure calls. Each newly added *CHR* constraint starts a search for matching rules. Executing a *built-in* constraint initiates similar searches. New in CHR$^{\neg}$ is that rules also trigger at the *removal* of CHR constraints. As with a procedure, when a rule fires, other constraints (its body) might be executed, and the execution does not return to the current active constraint *until* these calls have finished. Not surprisingly, this approach is used exactly because it corresponds closely to that of the stack-based programming languages to which CHR is compiled.

The refined semantics $\omega_r^{\neg}$ fixes the order in which searches for matching rules are conducted. That is why elements on the execution stack can become "occurrenced". Formally, $\mathbb{A}$ is a sequence of constraints, (occurrenced) identified CHR constraints and (occurrenced) negated CHR constraints, with a strict ordering, in which the top-most constraint is called the *active constraint*. The *positive* occurrences of constraints in the heads of the rules are numbered in a top-down, right-to-left manner (cf. [11]). 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.

In $\omega_r^{\neg}$ the stack can also contain *negative CHR constraints* $\neg c$ (we extend this notation to sequences in the usual way). Unlike in the positive case an active negative constraint will never be matched with a negative *occurrence*[2]. In other words, a removal, or the built-in store, triggers *rules*, not individual occurrences. An occurrenced negative CHR constraint $\neg c : j$ will therefore denote that only *the j-th rule* (in textual order) *where c occurs negatively* should be considered.

Example 17. Recall from Example 15 the CHR$^{\neg}$ pattern to maintain the minimum of a collection. These three CHR$^{\neg}$ rules rely on rule order to renew the minimum after the current minimal element is removed: *first* the old minimum has to be removed, *before* the *next* rule adds the new minimum. Both rules trigger on the *same* removal. This is a very natural way to read rules.

[2] The reason is that negative constraints do not always represent removed constraints, they are also put on the stack by the **Solve** transition (cf. Figure 3).

3.2.2 Transition rules

Given an initial goal $\mathbb{G}$ (a sequence), the *initial execution state* σ_0 is $\langle \mathbb{G}, \emptyset, true, \emptyset \rangle_1$ as in $\omega_t^\neg$. Again, execution proceeds by exhaustively applying transitions to σ_0, until the built-in solver state is unsatisfiable or no transitions are applicable. The transition rules of $\omega_r^\neg$ are listed in Figure 3.

Analogously to $\omega_t^{\neg 1}$, we define the fire-once refined operational semantics $\omega_r^{\neg 1}$ by removing the **AllowReapply** transition from $\omega_r^\neg$. Again, $\omega_r^{\neg 1}$ is a pure generalization of ω_r. It is straightforward to adapt the results of [11] to show that $\omega_r^\neg$ is an instance of $\omega_t^\neg$ and $\omega_r^{\neg 1}$ is an instance of $\omega_t^{\neg 1}$.

3.2.3 Determinism

Analogous to ω_r, $\omega_r^\neg$ only has two sources of non-determinism: the order in which the constraints are added to $\mathbb{A}$ in the **Solve**[3] transition and the choice of which partner constraints to use in the matchings of **Propagate**, **Simplify** and **TriggerNegative**. Note that if we would not require the **AllowReapply** transition to fire *immediately after it becomes applicable*, this would also become a potential source of non-determinism.

Because $\omega_r^\neg$ does not introduce any new form of non-determinism, it is again straightforward to extend the confluence results of [11] from ω_r to $\omega_r^\neg$.

3.3 Example derivation

Consider once more the CHR$^\neg$ program of Example 15. Suppose we add a simple rule (at the bottom of the program) that removes a given element:

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

Say the initial goal is `c(2), c(1), rm(1)`. Then, if we name the three original rules $\mathtt{r}_1$ to $\mathtt{r}_3$, the resulting derivation looks like ($\mathbb{B}$ is not included for brevity):

$\langle [\mathtt{c}(2), \mathtt{c}(1), \mathtt{rm}(1)], \emptyset, \emptyset \rangle_1$
$\quad \rightarrowtail_{act} \langle [\mathtt{c}(2)\#1 : 1, \mathtt{c}(1), \mathtt{rm}(1)], \{\mathtt{c}(2)\#1\}, \emptyset \rangle_2$
$\quad \rightarrowtail_{prop} \langle [\mathtt{min}(2), \mathtt{c}(2)\#1 : 1, \mathtt{c}(1), \mathtt{rm}(1)], \{\mathtt{c}(2)\#1\}, \{[1, \mathtt{r}_2]\} \rangle_2$
$\quad \rightarrowtail_{act} \langle [\mathtt{min}(2)\#2 : 1, \mathtt{c}(2)\#1 : 1, \mathtt{c}(1), \mathtt{rm}(1)], \{\mathtt{c}(2)\#1, \mathtt{min}(2)\#2\}, \{[1, \mathtt{r}_2]\} \rangle_3$
$\quad \rightarrowtail^* \langle [\mathtt{c}(2)\#1 : 1, \mathtt{c}(1), \mathtt{rm}(1)], \{\mathtt{c}(2)\#1, \mathtt{min}(2)\#2\}, \{[1, \mathtt{r}_2]\} \rangle_3$
$\quad \rightarrowtail^* \langle [\mathtt{c}(1)\#3 : 1, \mathtt{rm}(1)], \{\mathtt{c}(2)\#1, \mathtt{min}(2)\#2, \mathtt{c}(1)\#3\}, \{[1, \mathtt{r}_2]\} \rangle_4$
$\quad \rightarrowtail_{ara} \langle [\mathtt{c}(1)\#3 : 1, \mathtt{rm}(1)], \{\mathtt{c}(2)\#1, \mathtt{min}(2)\#2, \mathtt{c}(1)\#3\}, \emptyset \rangle_4$ ↩
$\quad \rightarrowtail_{prop} \langle [\mathtt{min}(1), \mathtt{c}(1)\#3 : 1, \mathtt{rm}(1)], \{\mathtt{c}(2)\#1, \mathtt{min}(2)\#2, \mathtt{c}(1)\#3\}, \{[3, \mathtt{r}_2]\} \rangle_4$
$\quad \rightarrowtail^* \langle [\mathtt{min}(1)\#4 : 3, \ldots], \{\mathtt{c}(2)\#1, \mathtt{min}(2)\#2, \mathtt{c}(1)\#3, \mathtt{min}(1)\#4\}, \{[3, \mathtt{r}_2]\} \rangle_5$
$\quad \rightarrowtail_{prop} \langle [\neg\mathtt{min}, \mathtt{min}(1)\#4 : 3, \ldots], \{\mathtt{c}(2)\#1, \mathtt{c}(1)\#3, \mathtt{min}(1)\#4\}, \{[3, \mathtt{r}_2], [4, 2, \mathtt{r}_3]\} \rangle_5$
$\quad \rightarrowtail_{ara} \langle [\neg\mathtt{min}, \mathtt{min}(1)\#4 : 3, \ldots], \{\mathtt{c}(2)\#1, \mathtt{c}(1)\#3, \mathtt{min}(1)\#4\}, \{[3, \mathtt{r}_2]\} \rangle_5$
$\quad$ (we now jump to a state little after the `rm(1)` constraint has removed `c(1)`...)
$\quad \rightarrowtail^* \langle [\neg\mathtt{c} : 1, \ldots], \{\mathtt{c}(2)\#1, \mathtt{min}(1)\#4\}, \emptyset \rangle_6$
$\quad \rightarrowtail_{trig} \langle [\neg\mathtt{min}, \neg\mathtt{c} : 1, \ldots], \{\mathtt{c}(2)\#1\}, \emptyset \rangle_6$
$\quad \rightarrowtail^* \langle [\neg\mathtt{c} : 1, \ldots], \{\mathtt{c}(2)\#1\}, \emptyset \rangle_6$
$\quad \rightarrowtail_{def} \langle [\neg\mathtt{c} : 2, \ldots], \{\mathtt{c}(2)\#1\}, \emptyset \rangle_6$
$\quad \rightarrowtail_{trig} \langle [\mathtt{min}(2), \neg\mathtt{c} : 2, \ldots], \{\mathtt{c}(2)\#1\}, \{[1, \mathtt{r}_2]\} \rangle_6$ ↩
$\quad \rightarrowtail^* \langle [], \{\mathtt{c}(2)\#1, \mathtt{min}(2)\#6\}, \{[1, \mathtt{r}_2]\} \rangle_7$

The derivation starts by activating `c(2)`, which fires the first rule. This adds `min(2)`, first on the stack, then also to the constraint store by an **Activate** (**Positive** or **Negative**). Note that firing $\mathtt{r}_1$ has added a tuple to the propagation history. The derivation continues with a series of **Default**s, followed by a **Drop** of `min(2)`.

[3] The definition of **Solve** we used is very weak. In practice, implementations should strive only to wake constraints that may potentially cause a rule to fire.

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 $\forall \mathbb{A}'' : \langle \mathbb{A}'', \mathbb{S}, \mathbb{B}, \mathbb{T}' \rangle_n \not\rightarrowtail \langle \mathbb{A}', \mathbb{S}', \mathbb{B}', \mathbb{T} \rangle_n$. This transition has to fire *whenever possible*.

---

1. Solve $\langle [c|\mathbb{A}], S_0 \sqcup S_1, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail \langle \{[x, \neg x] | x \in S_1\} +\!\!+ \mathbb{A}, S_0 \sqcup S_1, c \wedge \mathbb{B}, \mathbb{T} \rangle_n$ where c 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 *(which has never been active)*.

---

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 Negative $\langle [\neg c|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \rightarrowtail \langle [\neg c : 1|\mathbb{A}], \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}$

---

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 negative 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 the CHR predicate of c is d in a (renamed apart) rule r:

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

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 the CHR predicate of c is d in a (renamed apart) rule r:

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

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

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

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 negative ($\neg c$)).

Fig. 3. The transition rules of the refined operational semantics $\omega_r^\neg$ for CHR$^\neg$

The same happens for c(2), and the next constraint c(1) gets activated. Because this inserted c(1) into the store, rule $\mathtt{r}_1$ is no longer applicable with c(2), so **AllowReapply** removes the corresponding history tuple. Next, $\mathtt{r}_1$ fires once more and adds a new min/1 constraint. This causes r_3 to remove the old minimum, min(2)#2, from the store, also adding ¬min on top of the stack. Because negated constraints will not be matched, the constraint symbol alone is sufficient. An **AllowReapply** removes the newly added history tuple. Obviously, this will *always* happen immediately after the application of a simplification or simpagation rule.

We now make quite a big jump to the first time a removal triggers a rule. Because c(1) is removed, min(1) will be removed by rule r_1. This removal itself triggers nothing, but it is the $\neg c$ that, after a **Default** transition, will also trigger the next rule, r_2, to add the new minimum. Recall from Example 17 how we rely on *rule order* here! Finally, the new minimum is activated and added to store, and the derivation reaches it *final state*.

One final, crucial remark: the reason r_2 was capable of re-adding min(2) when 2 became the minimum again (i.e. firing a *second* time with c(2)#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 is a perfect example of why the **AllowReapply** transition is necessary!

4 Discussion: why not CHR$^{\neg}$?

In this section we focus on the disadvantages and limitations of the semantics introduced in the previous section.

4.1 Lost properties

Owing to the tradition of (constraint) logic programming, CHR features — besides a well-defined operational semantics [11] — a *declarative semantics*, i.e. CHR rules can be read as formulae in either classical first-order predicate logic [15], or linear logic [10]. This useful *theoretical* property is lost for CHR$^{\neg}$: negation as absence is an *operational* concept that cannot be integrated in *monotonic* logics (much like negation as failure in logic programming languages). Interesting research tracks would be to look for a declarative semantics of CHR$^{\neg}$ in a non-monotonic logic [8] or e.g. transaction logic [21].

A related, fundamental property of the CHR language is monotonicity [7]: rule application is independent of context. In regular CHR, adding constraints to a state cannot inhibit the applicability of a rule. Simply put, the monotonicity property can be states as follows: "if $A \rightsquigarrow B$, then $A \wedge C \rightsquigarrow B \wedge C$ for any C".

Clearly, the monotonicity property is lost in CHR$^{\neg}$, and with it all results that rely on it, most notably: the theoretical results on confluence [7, 6] of CHR programs under the ω_t semantics, and the corresponding confluence test. These are based on the fact that, given the monotonicity property, a so-called *minimal state* can be extended to any context, i.e. every other state where a rule is applicable contains its minimal state.

Also relying on monotonicity are the notions of parallelism introduced in [16][4].

[4] Note that [16] also assumes rule applications to be instantaneous, i.e. the removal and addition of constraints caused by the application of a rule are treated as an atomic action. It is interesting to note that we come to a similar conclusion in section 4.3.

4.2 Active versus passive negated heads

The $\omega_r^\neg$ semantics is shaped by the decision to allow negated heads to become *active*: rules will trigger if constraints are removed, or if negated guards become falsified. On the other hand negated heads also serve as an additional (*passive*) test of abscence in the applicability condition defined on page 9. We made a similar distinction when we introduced CHR$^\neg$ in Section 2. It can be argued that these are in fact two separate aspects, which should not be heedlessly mixed. On the other hand, we find a symmetric duality with positive occurrences: they are used to match the *active* constraint on addition, plus they constitute checks for the presence of partner constraints.

Either way, in practice we often want the negative heads to be and remain *passive*; i.e. we *only* want to consider a rule if it matches the active *positive* constraint, and use the negated heads simply to inspect the constraint store for the absence at those particular points in the execution.

Example 18. 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 clients who do not yet have this type of account. So, they add the following CHR$^\neg$ rule:

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

which works fine, except for the following: if some client already had an account, but closes it, the removal of his account will trigger the above rule, which may not be intended.

As a solution, we propose a new pragma `passive`, similar to its generally accepted counterpart for positive occurrences [5, 32, 29]. In general, a pragma is defined in computer science as:

> a compiler directive embedded in source code by programmers, communicating additional "pragmatic" information (on control, optimizations, ...)

Positive ***pragma passive***'s declare that a certain positive *occurrence* should not be considered (in other words: we add, to the **Simplify** and **Propagate** transition rules of $\omega_r^\neg$, the extra condition that the j'th occurrence must not be declared passive). Because negated constraints in $\omega_r^\neg$ work on a per-rule base, it is however pointless to declare *individual* negative occurrences passive. Instead a *negative* ***pragma passive*** declares that a constraint should never become active for a certain rule (i.e. we add a straightforward condition to the **TriggerNegative** rule).

Example 19. The rule of Example 18 then becomes:

```
send @ client(X) \\ account(X,_)
     ==> send_brochure_to(X) pragma passive(account/2).
```

The use of such pragmas makes a program less declarative. One important difference between the positive and negative pragma passive is that the former should *not* be used to alter the semantics of the program (i.e. only to aid the compiler to detect passive occurrences, needed for certain optimizations[5]), while the latter clearly *are* intended to alter semantics (i.e. they are part of the control).

[5] Declaring a positive occurrence passive that could become active can result in unexpected results in most optimizing CHR implementations.

4.3 Problems of $\omega_r^{\neg}$

Example 20 (Destructive update). Reconsider example 18, which had the following rule (the `pragma passive` is irrelevant for the discussion here):

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

Suppose the CHR$^{\neg}$ program of the bank also contains a rule to handle deposits:

```
deposit(X,Amount), account(X,B) <=> account(X,B + Amount).
```

The latter 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)`) will trigger rules (`send`), even though the constraint will immediately be replaced with an updated version. In our example 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.

In this case, it would help to allow the programmer to explicitly choose the moment a constraint is removed, say something like:

```
deposit(X,A), account(X,B)#Id <=> account(X,B+A), chr_remove(Id).
```

However, this clearly impairs the declarativeness of the rule. Besides, we will now have an intermediate state where both the old and the updated constraint are in the constraint store. This might just as well cause erroneous results.

Example 21 (Order). In the presence of negation, the order in which constraints are added becomes more important. The first rule of the following program will, at birth, cause each child to be declared orphan by the second rule.

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

The solution here is to rearrange the constraints in the body of the first rule. Another example can be found in the graph program listed in Figure 1, where nodes have to be created before edges. There are also programs where 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`)`. 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 22 (Atomicity). Suppose we want to improve the graph program of Figure 1 by adding

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

as the first rule of the program. The idea is to allow the creation of edges without the extra burden of adding node constraints first (cf. example 21). Unfortunately, this innocent-looking rule causes unexpected behavior. Consider the query "`edge(a,b), edge(b,c)`". Given the new rule, the first constraint propagates `node(a)` and `node(b)`. 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, and the `set_sem_nodes` removes it immediately. This removal however triggers the `missing_to` rule, which deletes `edge(b,c)`. Remember, there is no `node(c)` yet! So the final constraint store will model the broken graph `edge(a,b)`, `node(a)`, `node(b)`, `node(c)`, with the edge `edge(b,c)` missing. This problem can be solved in many ways, for example by replacing our new rule with:

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

Here the `passive` pragmas are needed to 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*, but the addition of constraints remains *incremental*.
- 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 examples in this section — with the apparent exception of the above second item — are symptoms of the same, fundamental problem of integrating negation with CHR while preserving the conventional operational semantics [11]. In this semantics a rule is essentially executed in a series of successive, discrete steps. Constraints (from the query or the body of a rule) 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, and so on. This is a known issue with the common operational semantics of regular CHR, but it hardly ever poses a problem in practice. But as the examples in this section clearly show, the integration of negation severely exacerbates the problem.

4.3.1 Potential solutions

One solution would be to make rule application (more) atomic. This however becomes complicated if the rules contain host-language statements: the slightly more procedural flavor of CHR is exactly what makes it possible to (efficiently) compile CHR to several different host-languages.

Additional pragmas can be added to avoid some of the above problems:

Example 23. Consider the problem described in Example 22. We could disallow triggering caused by the removal in the set semantics rule for nodes by adding a pragma to this set semantics rule:

```
node(A) \ node(A)#Id <=> true pragma no_remove_trigger(Id).
```

meaning that if a `node/1` constraint is removed by this rule, this does not trigger the rules in which the removed constraint occurs negatively.

We consider this approach of adding pragmas a not very desirable ad-hoc solution which is somewhat contrary to the declarative nature of CHR.

A more fundamental solution would be to have a significantly different operational semantics for CHR (and CHR$^\neg$), for instance a more breadth-first batch-like semantics or a semantics in which the rule order is enforced in a stronger way than in ω_r. At this point there has been little interest in the topic of alternative operational semantics for CHR, but it seems to be a promising area of future research.

5 Transforming CHR$^\neg$ to CHR

In this section we present a source-to-source translation scheme T from CHR$^\neg$ to CHR. It forms the basis for a prototype CHR$^\neg$ implementation in Prolog, which has enabled us to experiment with CHR$^\neg$ and to gain insight in its subtleties. This section should also be a clear illustration of the fact that negation as absence is more than some simple syntactic sugar: the full transformation is relatively involved. A CHR$^\neg$ program of r rules and c user-defined constraints requires up to $\mathcal{O}(rc)$ rules when transformed to regular CHR.

An arbitrary CHR$^\neg$ program is transformed in a series of successive steps:

- T_i ensures that each CHR constraint receives an *explicit* constraint identifier (as an extra argument).
- T_h uses negation to add explicit propagation history management.
- T_n eliminates all negated heads.

The complete transformation is then $T = T_n \circ T_h \circ T_i$. The goal is that, for an arbitrary CHR$^\neg$ program P, executing $T(P)$ under ω_r semantics is equivalent with executing the original program P under $\omega_r^\neg$.

5.1 Explicit identifiers

First, T_i adds a rule of the form

$$c(\bar{X}) \iff c_id(_Id, \bar{X})$$

for each CHR constraint c, so every instance of a constraint c is replaced with a uniquely identified constraint c_id. We use fresh, unbound variables as *explicit identifiers*. These identifiers are similar to the *implicit identifier* assigned to each constraint in the formal semantics of Section 3. They allow both T_h and T_n to distinguish between otherwise identical CHR constraints.

We now also have to adapt the rules of the original program to use the identified versions of the constraints. It suffices to transform the heads (both positive and negative): T_i replaces all occurrences (of the form $c(\bar{X})$) with a corresponding, explicitly identified occurrence (i.e. $c_id(Id, \bar{X})$, with Id a fresh variable).

Example 24. The CHR$^\neg$ program from Example 15 becomes:

```
c(X) <=> c(_Id, X).
min(X) <=> min(_Id, X).

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

It should be clear that T_i never affects a derivation[6] under $\omega_r^\neg$.

5.2 Explicit histories

The implementation of the **AllowReapply** transition requires the removal of propagation history tuples. Also, T_n transforms a single CHR$^\neg$ rule in multiple CHR rules. These rules then have to somehow *share* the same propagation history.

[6] Only the real purist will critique that it *can* influence a derivation *in theory*, due to the weak definition of the **Solve** transition. It is however clear that it will never happen in practice.

The propagation history is kept *implicitly* by the CHR runtime system. Adding low-level support to accomplish the above goals would be one option, but for our exploratory transformation scheme we preferred to minimize the changes required to the underlying CHR system: propagation histories are maintained *explicitly* using CHR$^\neg$ constraints. To do this we need a way to uniquely identify constraints. Let $id_e(c)$ denote the function that maps a constraint to its unique identifier, i.e. to its first argument (keep in mind we are working on the result of T_i), and extend it to a conjunction of constraints in the straightforward manner. Then T_h replaces each *propagation* rule r of the form (1) with:

$$r \ @ \ H_1 \setminus\setminus N_1 \mid G_1 \setminus\setminus \ \ldots \ \setminus\setminus N_k \mid G_k \setminus\setminus hist_r(id_e(H_1)) \\ \Longrightarrow G \mid hist_r(id_e(H_1)), B \ \texttt{pragma no_history,passive(hist_r)}$$

Example 25. The only propagation rule of Example 24 is replaced with:

```
c(Id1, X) \\ c(_, Y) | Y < X \\ hist_r4(Id1)
  ==> hist_r4(Id1), min(X) pragma no_history, passive(hist_r4/1).
```

This suffices for $\omega_r^{\neg 1}$, the fire-once semantics of CHR$^\neg$, which does not contain the **AllowReapply** transition rule from $\omega_r^\neg$. We know from Section 3 that this extra rule removes propagation history tuples to allow reapplication. We will contemplate its implemention shortly, but first we motivate the use of both pragmas. Note however that both are actually only necessary in the full semantics.

The first pragma, `no_history`, declares that the *implicit* propagation history of the annotated (propagation) rule does not have to be kept. The implicit history would otherwise always prevent reapplication, even if **AllowReapply** has removed the *explicit* history tuples. This extra pragma is the *only* new requirement we need from a CHR runtime to be able to run the transformed code. This pragma should be very easy to implement.

Concerning the second pragma, it is easy to see that triggering a rule at the removal of a history tuple would not make much sense: after all, a tuple is only removed by **AllowReapply** if a rule is *not* applicable for a certain matching. Nonetheless, the pragma must not be discarded: we know from the discussion in Section 4 that a removal can trigger a negated head that checks for the absence of constraints other the removed one. So, if a rule *is* applicable for certain combinations of constraints *other* then the one represented by the just removed tuple, the rule *will* fire after the triggering on its removal. This alters the execution order, and can influence the result of a derivation (if the program is non-confluent).

5.2.1 The AllowReapply transition

The semantics states that a *hist_r* constraint has to be removed from the store if a rule r can no longer fire with the same combination of positive heads. We distinguish two cases:

- The matching of the *positive* head that caused the rule to fire is no longer valid; using CHR$^\neg$ it suffices to add a rule of the form:

$$hist_r(id_e(H_1)) \setminus\setminus H_1 \mid G \Longleftrightarrow true \tag{2}$$

- A matching for one of the *negative* heads becomes present. So, for $1 \leqslant i \leqslant k$ we add:

$$H_1, N_i \setminus hist_r(id_e(H_1)) \Longleftrightarrow G, G_i \mid true \tag{3}$$

  The positive guard G only has to be included if it introduces existential variables used in G_i.

In Section 3 we motivated that each **AllowReapply** transition has to be applied as soon as it becomes applicable.

As a first attempt, we could try to add rules (2) and (3) in front of the rest of the program. This will make them fire before any other rule when, respectively, a constraint occurring positively is removed or a constraint occurring negatively is added.

Unfortunately, this is not enough. Both types of rules also rely on the triggering of guards[7]. The order in which constraints are reactivated however is not deterministically defined by ω_r^- (cf. page 12). This means that, unless prevented, several rules could trigger before all necessary history tuples are removed. It is possible to construct examples where this might affect the outcome.

A lesser-known CHR pattern to force some rule of the form (1) to fire before any other rule after a **Solve** transition consists of constructing rules of the form ($\bar{X}$ are fresh variables)

$$c(\bar{X}), H_1 \setminus H_2 \setminus\setminus N_1 \mid G_1 \setminus\setminus \ \ldots \ \setminus\setminus N_k \mid G_k \Longleftrightarrow G \mid C$$

for *each* CHR constraint c. All occurrences in H_1 and H_2 of these rules can be declared passive. In fact, if the original rule is a propagation rule, this becomes mandatory, but we will only use the pattern on simplification and simpagation rules here. In this specific case we also only have to consider all *c_id* constraints[8], introduced by T_i. These rules then have to be added, together with the original rule, in front of the rest of the program.

When we apply this pattern to all rules (2) and (3), the **AllowReapply** transition will be tried first, not only at constraint addition or removal, but also after each **Solve** transition, as is required. Note from the example below that all *extra* negative occurrences (i.e. all but those from the original rule (2)), can be made passive.

Example 26. The result of our running example (cf. Examples 24 and 25) after the first two transformation steps, T_i and T_h, is:

```
hist_r4(Id1) \\ c(Id1, _) <=> true.
c(_, _) \ hist_r4(Id1) \\ c(Id1, _) <=> true pragma passive(c/2).
min(_, _) \ hist_r4(Id1) \\ c(Id1, _) <=> true pragma passive(c/2).

c(Id1, X), c(_, Y) \ hist_r4(Id1) <=> Y < X | true.
c(_, _), c(Id1, X), c(_, Y) \ hist_r4(Id1) <=> Y < X | true.
min(_, _), c(Id1, X), c(_, Y) \ hist_r4(Id1) <=> Y < X | true.

c(X) <=> c(_Id, X).
min(X) <=> min(_Id, X).

min(_, X) \\ c(_, X) <=> true.
c(Id1, X) \\ c(_, Y) | Y < X \\ hist_r4(Id1)
  ==> hist_r4(Id1), min(X) pragma no_history, passive(hist_r4/1).
min(_, X) \ min(_, Y) <=> X ≤ Y | true.
```

[7] For the positive case (2) this is only an issue if the guard is non-monotonic (cf. page 8).

[8] Several other optimizations are possible, like excluding never-stored constraints, constraints with only ground arguments, ... This is however beyond the scope of our exploratory transformation scheme.

5.3 Eliminating negated heads

The transformation T_n eliminates negated heads, which is of course the principal goal of the transformation. We replace every rule of a program P with a sequence of rules. In the remainder of this section we will consider one arbitrary rule r of a program P, and construct the corresponding rules in the transformed program. We can consider this rule to be of the form (1), with possibly some pragmas[9] introduced by T_h. We will use $\bar{p}$ to denote the variables occurring in the positive heads and guard, i.e. $\bar{p} = vars(H_1 \wedge H_2 \wedge G)$.

For each constraint $c \in H_2$ that is *removed* by rule r, we have to reconsider *all* rules r_i of P (including possibly r itself) where c occurs negatively. This is done using `trigger_`r_i constraints. Before we can give the actual rules replacing r we have to introduce some more formal notation.

Assume H_2 is of the form $c_1, \ldots, c_n$. Let $\sigma(c)$ return the constraint symbol of a constraint c, and $\Sigma(r_i) = \{\sigma(c) \mid c \text{ occurs negatively in rule } r_i\}$. We also have to take into consideration the pragma passives: we define

$$\Pi(r_i) = \{c \mid r_i \text{ is annotated with } \texttt{pragma passive}(c)\}$$

We define some auxiliary notation $may_trigger^{r_i}$ as follows:

$$may_trigger^{r_i} = \begin{array}{ll} \texttt{trigger_}r_i & \text{if } \sigma(c_j) \in \Sigma(r_i) \backslash \Pi(r_i) \\ \texttt{true} & \text{otherwise} \end{array}$$

Finally, we define the symbol $removed^{c_j}$ as follows:

$$removed^{c_j} = (may_trigger^{r_1}, \ldots, may_trigger^{r_n})$$

Intuitively, $removed^{c_j}$ corresponds to $\neg c_j$ in $\overleftarrow{\omega_r}$, that is, it causes all rules in which c_j occurs negatively to be tried.

Rule r is then replaced with:

$$H_1 \setminus H_2 \Longleftrightarrow G, negation_r(\bar{p}) \mid removed^{c_1}, \ldots, removed^{c_n}, B$$

If $k > 0$, we also have to be able to activate the rule r upon removal of a negatively occurring constraint, so we add:

$$\texttt{trigger_}r, H_1 \setminus H_2 \Longleftrightarrow G, negation_r(\bar{p}) \mid removed^{c_1}, \ldots, removed^{c_n}, B$$
$$\texttt{trigger_}r \Longleftrightarrow true.$$

In the presence of non-monotonic guards this is not sufficient: a rule also has to fire if one of the negated guards is no longer entailed by the built-in constraint store. So we also add, for each negated head N_i, a rule of the form:

$$H_1, N_i \setminus H_2 \Longleftrightarrow G, negation_r(\bar{p}) \mid removed^{c_1}, \ldots, removed^{c_n}, B$$

The rules replacing the original rule no longer contain negative heads. Instead, they all have an extra conjunct in the guard, $negation_r(\bar{p})$. This CHR constraint has to implement the (passive) check for absence of matching constraints. To be precise, for each $i \in \{1, \ldots, k\}$ we add a rule:

$$N_i \setminus negation_r(\bar{p}) \Longleftrightarrow G_i, disjunct(id_e(H_1, H_2), id_e(N_i)) \mid fail.$$

This rule states that $negation_r(\bar{p})$ will fail as soon as one of the negated heads matches a set of constraints in the store, but only if these constraints are different

[9] Our transformation ignores pragmas other than `no_history` and "negative" `passive`. Taking other pragmas into account as well is only a small extension.

from the ones used in the matching of the positive head. The latter condition is checked by the $disjunct(id_e(H_1, H_2), id_e(N_i))$ guard, which only succeeds if the two set of unique identifiers are disjunct. Note that these identifiers are included in $\bar{p}$ (T_i has been applied as a previous transformation step). We add the following rule *after* the k previous rules, stating that $negation_r(\bar{p})$ succeeds if no negated head matches:

$$negation_r(\bar{p}) \iff true.$$

There is one special case when N_i is of the form $hist_r(id_e(H_1))$ (cf. T_h), where the $hist_r$ constraint does not have an explicit identifier. $hist_r$ cannot occur positively though, so we can simply omit the extra $disjunct$ conjunct from the guard. Other constraints without an explicit identifier (the original constraints and some other auxiliary constraints introduced by T_h) never occur negatively, so they do not pose a problem.

This concludes the T_n transformation step.

The attentive reader will have noticed however that we used CHR constraints in the guard of several rules. Most specifications of CHR however only allow built-in (ask) constraints in a guard. CHR constraints are excluded because they generally are tell constraints[10] that could alter the execution state. The CHR constraints we used in guards are all of the form $negation_r$. They are never-stored constraints that can be considered to be ask-constraints in the sense that they only perform a *query*-operation on the constraint store. As a result, there is no reason to forbid the use of $negation_r$ constraints in the guards of rules — and it can be done without any modification to the underlying CHR implementation.

Example 27. Figure 4 lists the result of applying the full transformation ($T_n \circ T_h \circ T_i$) to the running example, extended with the extra rule first introduced in Section 3.3:

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

6 Related work

Ever since the first generation of production (rule) systems [12, 3], negation similar to the one we introduced into CHR has always played an important role. In fact, this was one of the main motivations for us to investigate negation as absence in the context of CHR. In this section we give a short overview of some aspects, related to our work here, of these rule based languages.

OPS5 [12] offered 'negative condition elements', corresponding to a limited form of negated heads: they were only allowed to have one conjunct.

In CLIPS [3] you can use the logical connectives `not`, `and` and `or` on the left-hand side of a rule. These connectives can be nested arbitrarily. It is clear that equivalents of our negated heads can be accomplished using only `not` and `and`. Disjunctions are interpreted as nothing more than syntactic sugar, which could be easily added to CHR as well: each left-hand side is simply rearranged (using distributivity and De Morgan's second law) in such a way that all `or` connectives become top-level; each of the resulting disjuncts then gives rise to an extra rule (with the same body).

The main difference with CHR$^\neg$ is thus that negated conjunctions can be nested inside other negated conjunctions. In fact, CLIPS introduced syntactic sugar for two interesting patterns that can be created this way:

[10] [25] shows how a large class of CHR constraints can be turned into ask constraints, making it safe to use them in a guard.

minimum.chr

```
%========== AllowReapply rules (+ priority pattern) ===============%
%---------  a) 'history GC' (positive head gone)  ----------%
hist_r4(Id1) <=> negation_r1(Id1) | true.
c(Id1,_) \ negation_r1(Id1) <=> fail.
negation_r1(_) <=> true.
trigger_r1 \ hist_r4(Id1) <=> negation_r1(Id1) | true.
trigger_r1 <=> true.

% [+ 9 additional rules to enforce priority of the above]

%---------  b) 'real' AllowReapply (negated head added)  ---%
c(Id1,X), c(_,Y) \ hist_r4(Id1) <=> Y < X | true.
% priority rules for the above:
c(_,_), c(Id1,X), c(_,Y) \ hist_r4(Id1) <=> Y < X | true.
min(_,_), c(Id1,X), c(_,Y) \ hist_r4(Id1) <=> Y < X | true.
rm(_,_), c(Id1,X), c(_,Y) \ hist_r4(Id1) <=> Y < X | true.


%========== Explicit identifiers ==================================%
c(X) <=> c(_Id,X).
min(X) <=> min(_Id,X).


%========== Transformation of the original rules ==================%
%---------  min(X) \\ c(X) <=> true.  ----------------------%
min(_,X) <=> negation_r11(X) | true.
c(_,X) \ negation_r11(X) <=> fail.
negation_r11(_) <=> true.
trigger_r11 \ min(_,X) <=> negation_r11(X) | true.
trigger_r11 <=> true.

%---------  c(X) \\ c(Y) | Y < X ==> min(X).  --------------%
c(Id1,X) ==> negation_r12(Id1,X) | hist_r4(Id1), min(X) pragma no_history.
c(Idc,Y) \ negation_r12(Id1,X) <=> Y < X, Idc \== Id1 | fail.
hist_r4(Id1) \ negation_r12(Id1,_) <=> fail.
negation_r12(Id1,X) <=> true.
trigger_r12, c(Id1,X) ==> negation_r12(Id1,X) | hist_r4(Id1), min(X)
                                                    pragma no_history.
trigger_r12 <=> true.

%---------  min(X) \ min(Y) <=> X ≤ Y | true.  ------------%
min(_,X) \ min(_,Y) <=> X ≤ Y | true.

%---------  rm(X), c(X) <=> true.  -------------------------%
rm(_,X), c(_,X) <=> trigger_r1, trigger_r11, trigger_r12.
```

Fig. 4. The result of applying the full transformation ($T_n \circ T_h \circ T_i$) to the CHR$^\neg$ program introduced in Example 15.

– `(exists (X))` is defined as `not (not (X))`. Note that negation is not involutive: if a precondition `X` is surrounded by a double negation, the rule will not fire for a new combination satisfying the precondition if the rule was already applied with another such combination.
– `(forall (X) (Y))` is equivalent with as `(not (and (X) (not (Y))))`

In general, arbitrary combinations of connectives can be constructed this way. Note that, as [2] states:

> Care must be taken when combining `not` [Conditional Elements] with `or` and `and` CEs; the results are not always obvious!

Even though negation does help to express some limited instances of these rule antecedents, expressing the more general cases in CHR$^\neg$ (and thus CHR) requires constructions similar to the ones we used to transform CHR$^\neg$ to CHR in Section 5.

Most modern production systems can be considered to be direct descendants of these early systems. One of the best known rule engines for Java for example, Jess [14], can be regarded as a Java-oriented clone of the CLIPS language. Drools [4] is the only open-source rule engine we know of that has become very popular without support for negation. Drools 3.0 (or JBoss Rules [20] as it is also called now) however will introduce logical connectives similar to the ones used by CLIPS and Jess. We do not have much information about the dozens of commercial implementations [18, 19, 28], but we expect them to have similar features.

So far we have only compared the systems in terms of syntactical expressiveness. Comparing the actual semantics is complicated by the fact that, to the best of our knowledge, none of these systems have ever formally specified their concrete operational semantics. From experimental evaluation though, we know that the semantics formalized in Section 3 is a very close approximation.

The reason these systems do not suffer from the issues discussed in Section 4.3 is the different execution mechanism. Almost all current production (rule) systems are based on the RETE [13] algorithm, in which rules are executed in a more atomic fashion: the next rule is only determined after *all* necessary constraints (facts) are added and removed.

LEAPS [9] is an asymptotically better [23, 22] execution algorithm for production rules, and is in many ways much closer to the way CHR is commonly compiled then RETE is. One of the differences nonetheless is that in LEAPS rule application again is an atomic action: it is only *after* all constraints (facts) are added and removed, that the next active constraint (dominant object) is selected. The only production rule system we know of that implements LEAPS is Venus [24] (and Drools 3.0 as an experimental alternative to RETE).

So when comparing production rules with CHR, the former feature more expressiveness in the left-hand sides of their rules. CHR$^\neg$ is a first step towards overcoming this shortcoming of CHR. However, the non-atomic rule application of the default operational semantics of CHR (and CHR$^\neg$) is problematic, as discussed in Section 4. On the other hand, the performance of CHR$^\neg$ is potentially superior to that of most production rule systems because of its efficient evaluation algorithm (similar to LEAPS) and because CHR$^\neg$ programs are compiled while production rules are usually interpreted. A final advantage of CHR$^\neg$ over production rule systems is its clearly specified operational semantics, which is lacking in production rule systems.

7 Conclusion

We proposed CHR$^\neg$, an extension of CHR that adds negation to the CHR language: CHR$^\neg$ rules can test for the *absence* of constraints and can fire after the *removal* of a constraint. Our main contribution is the investigation of the benefits and issues of integrating this form of negation with CHR. We determined an operational semantics for CHR$^\neg$ and formalized it as a minimal extension of the conventional CHR semantics. We presented several alternatives, and motivated the choices we made extensively by examples. We designed a transformation from CHR$^\neg$ to CHR and used it to implement a prototype implementation.

Other, related languages (e.g. production rules) offer similar syntactical features. To the best of our knowledge, we are the first to examine negation in the context of CHR.

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 CHR$^\neg$. The issues discussed in Section 4 also indicate that programming in CHR$^\neg$ might require a thourough understanding of the execution mechanism, which 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 to the language in the sense that constraint insertion and constraint removal become equally important. We expect CHR$^\neg$ to have interesting but complicated theoretical and practical properties, which can only be uncovered by further research.

7.1 Future work

We intend to formally proof the correctness and completeness of the transformation scheme presented in Section 5. If we decide to implement CHR$^\neg$ in the K.U.Leuven CHR compilers [26, 30], there is a wide spectrum of possible optimizations. An investigation of a more breadth-first operational semantics of CHR, with more atomic rule application, may very well be warranted first (cf. Section 4). It might also be interesting to look into the added expressiveness of other extensions of the left-hand side of rules (disjunction, nested logical operators, ...) and the expected difficulties of their integration with CHR.

Acknowledgements We would like to 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.

References

1. Information Technology – Programming Language – Prolog – Part 1: General Core. ISO/IEC 13211-1, 1995.
2. *CLIPS 6.23 Reference Manual, Volume 1 – Basic Programming Guide*, June 2005.
3. CLIPS – A Tool for Building Expert Systems. http://www.ghg.net/clips/CLIPS.html, May 2006.
4. The Drools home page. http://drools.codehaus.org/, May 2006.
5. *The SICStus Prolog User Manual*, chapter on Constraint Handling Rules. 3.12.5 edition, March 2006.
6. Slim Abdennadher. Operational semantics and confluence of constraint propagation rules. In Gert Smolka, editor, *Proceedings of Third International Conference on Principles and Practice of Constraint Programming*, pages 252–266, Schloss Hagenberg, Austria, 1997. Springer LNCS.
7. Slim Abdennadher, Thom Frühwirth, and Holger Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal*, 4(2):133–165, 1999.

8. G. Aldo Antonelli. Non-monotonic logic. Stanford Encyclopedia of Philosophy, Spring 2006. http://plato.stanford.edu/archives/spr2006/entries/logic-nonmonotonic/.
9. Don Batory. The LEAPS algorithms. Technical Report CS-TR-94-28, University of Texas, Austin, TX, USA, 1994.
10. Hariolf Betz and Thom Frühwirth. A Linear-Logic Semantics for Constraint Handling Rules. In Peter van Beek, editor, *11th Conference on Principles and Practice of Constraint Programming (CP 2005)*, volume 3709 of *Lecture Notes in Computer Science*, pages 137–151, Sitges Spain, October 2005. Springer Verlag.
11. Gregory J. Duck, Peter J. Stuckey, María García de la Banda, and Christian Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *Proc. 20th Intl. Conference on Logic Programming (ICLP'04)*, pages 90–104, St-Malo, France, 2004.
12. Charles L. Forgey. OPS5 User's Manual. Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie Mellon University, Pittsburgh, July 1981.
13. Charles L. Forgey. RETE: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.
14. Ernest Friedman-Hill et al. Jess, the Rule Engine for the Java Platform. http://herzberg.ca.sandia.gov/jess/, May 2006.
15. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.
16. Thom Frühwirth. Parallelizing Union-Find in Constraint Handling Rules Using Confluence Analysis. In Maurizio Gabbrielli and Gopal Gupta, editors, *Proc. 21st Conference on Logic Programming (ICLP 2005)*, volume 3668 of *Lecture Notes in Computer Science*, pages 113–127, Sitges, Spain, October 2005. Springer Verlag.
17. Christian Holzbaur, Maria García de la Banda, Peter J. Stuckey, and Gregory J. Duck. Optimizing compilation of constraint handling rules in HAL. *Theory and Practice of Logic Programming (TPLP)*, 5(4-5):503–532, July 2005.
18. ILOG. ILOG rules. http://www.ilog.com/products/rules/, May 2006.
19. Fair Isaac. Blaze advisor rules management technology. http://www.fairisaac.com, May 2006.
20. JBoss. JBoss Rules. http://www.jboss.com/products/rules, May 2006.
21. Marc Meister. A Transaction Logic Semantics for CHR. Seminar Day on Constraint Handling Rules (http://www.cs.kuleuven.be/∼toms/CHR/chr_wog.html), May 2006.
22. Daniel Miranker. TREAT or RETE, neither, LEAPS is best. http://www.cs.utexas.edu/∼miranker/treator.htm, May 2006.
23. Daniel P. Miranker, David A. Brant, Bernie Lofaso, and David Gadbois. On the Performance of Lazy Matching in Production Systems. In *8th National Conference on Artificial Intelligence*, pages 685–692. AAAI, July 1990.
24. Daniel P. Miranker, L. Obermeyer, L. Warshaw, and J. C. Browne. Venus: An object-oriented extension of rule-based programming. Technical report, University of Texas, Austin, 1998.
25. Tom Schrijvers, Bart Demoen, Gregory Duck, Peter Stuckey, and Thom Frühwirth. Automatic Implication Checking for CHR Constraints. In H. Cirstea and N. Martí-Oliet, editors, *RULE'05: Proceedings of the 6th International Workshop on Rule-Based Programming*, Nara, Japan, April 2005.
26. Tom Schrijvers et al. The K.U.Leuven CHR system home page, May 2006. http://www.cs.kuleuven.ac.be/∼toms/Research/CHR/.
27. Jon Sneyers, Tom Schrijvers, and Bart Demoen. Dijkstra's Algorithm with Fibonacci Heaps: An Executable Description in CHR. In *20th Workshop on Logic Programming (WLP'06)*, Vienna, Austria, February 2006.
28. Haley Systems. Haley Rules. http://www.haley.com/, May 2006.
29. Peter Van Weert. The K.U.Leuven JCHR system home page. http://www.cs.kuleuven.be/∼petervw/JCHR, May 2006.
30. Peter Van Weert, Tom Schrijvers, and Bart Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In *Proc. 2nd Workshop on Constraint Handling Rules (CHR'05)*, pages 47–62, Sitges, Spain, October 2005.
31. Peter Van Weert, Jon Sneyers, Tom Schrijvers, and Bart Demoen. CHR$^{\neg}$: Constraint Handling Rules with negation. In *Proceedings of the 3rd Workshop on Constraint Handling Rules (CHR'06)*, Venice, Italy, July 2006. Submitted.
32. Jan Wielemaker, Tom Schrijvers, et al. *SWI-Prolog Reference Manual*, chapter 7 CHR: Constraint Handling Rules. 5.6.10 edition, 2006.