User-definable Rule Priorities for CHR

Leslie De Koninck * Tom Schrijvers † Bart Demoen

Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200a, 3001 Heverlee, Belgium
FirstName.LastName@cs.kuleuven.be

Abstract

This paper introduces CHR$^{\text{rp}}$: Constraint Handling Rules with user-definable rule priorities. CHR$^{\text{rp}}$ offers flexible execution control which is lacking in CHR. A formal operational semantics for the extended language is given and is shown to be an instance of the theoretical operational semantics of CHR. It is discussed how the CHR$^{\text{rp}}$ semantics influences confluence results. A translation scheme for CHR$^{\text{rp}}$ programs with static rule priorities into (regular) CHR is presented. The translation is proven correct and benchmark results are given. CHR$^{\text{rp}}$ is related to priority systems in other constraint programming and rule based languages.

Categories and Subject Descriptors D.1.6 [*Programming Techniques*]: Logic Programming; D.3.2 [*Programming Languages*]: Language Classifications—Constraint and Logic Languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control Structures

General Terms Languages

Keywords Constraint Handling Rules, Execution control, Rule priorities

1. Introduction

Constraint Handling Rules (CHR) [13] is a rule based language, originally designed for the implementation of constraint programming systems, but also increasingly used as a general purpose programming language [22, 28]. While language features such as multi-headed rules, guards and multi-set semantics make CHR very flexible as far as specifying the logic of a program is concerned, flexible execution control is lacking. On the one hand, the theoretical operational semantics (ω_t) of CHR is highly non-deterministic and allows for practically no execution control at all. On the other hand there is the refined operational semantics (ω_r) [10], used by current CHR implementations, which fixes most of the execution strategy, but which is low-level and procedural in nature. To change the default execution strategy in implementations based on the ω_r semantics, one has to use auxiliary constructs like flag constraints. Such an approach is neither flexible nor efficient.

In this paper, we extend CHR with user-definable rule priorities. This extension, called CHR$^{\text{rp}}$, offers the flexible execution control needed for implementing highly efficient Constraint (Logic) Programming systems. Moreover, it facilitates implementing rule-based algorithms as these often require certain rules to be tried before others, either for efficiency reasons, or for correctness. Rule priorities allow the programmer to write programs that are more concise and efficient, but potentially not confluent under the ω_t semantics. They support a high-level form of execution control that is more declarative, flexible and comprehensible compared to the control offered by the refined operational semantics. In CHR$^{\text{rp}}$, all execution control information is explicit in the rule priority annotations, which causes a clear distinction between the logic and the control aspect of a CHR$^{\text{rp}}$ program.

As an alternative to rule priorities, CHR could be extended by other forms of execution control, most notably constraint priorities and execution in phases. These alternatives are discussed in the accompanying report [8]. In [31] it is shown that any algorithm can be implemented in CHR preserving time and space complexity, so neither rule priorities nor any other form of execution control increases the computational power or improves the complexity of CHR. Rule priorities however do improve the expressivity of CHR and make programs more concise and adaptable.

1.1 Overview

The rest of this paper is organized as follows. Section 2 reviews the syntax and semantics of CHR. In Section 3 motivation for, and examples of CHR$^{\text{rp}}$ are given. The syntax and operational semantics of CHR$^{\text{rp}}$ is formalized in Section 4. In that section, we also discuss its theoretical properties. Then, in Section 5, it is shown how CHR$^{\text{rp}}$ programs with only static rule priorities, can be implemented as a source-to-source transformation to regular CHR, based on the refined operational semantics. In Section 6, we prove the correctness of our implementation, and investigate the runtime and space overhead it introduces. An empirical evaluation follows in Section 7 and Section 8 concludes.

2. Constraint Handling Rules

This section reviews the syntax and semantics of Constraint Handling Rules (CHR). For a more thorough introduction, see [13] or [25].

2.1 Syntax and Declarative Semantics

A constraint $c(t_1, \ldots, t_n)$ is an atom of predicate c/n with t_i a host language value (e.g., a Herbrand term in Prolog) for $1 \leq i \leq n$. There are two types of constraints: built-in constraints and CHR constraints (also called user-defined constraints). The CHR constraints are solved by the CHR program whereas the built-in

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

† Post-Doctoral Researcher of the Fund for Scientific Research - Flanders (Belgium) (F.W.O. - Vlaanderen).

PPDP'07 July 14–16, 2007, Wroclaw, Poland.

1. Solve $\langle \{c\} \uplus G, S, B, T\rangle_n \overset{\omega_t}{\mapsto}_P \langle G, S, c \wedge B, T\rangle_n$ where c is a built-in constraint.
2. Introduce $\langle \{c\} \uplus G, S, B, T\rangle_n \overset{\omega_t}{\mapsto}_P \langle G, \{c\#n\} \cup S, B, T\rangle_{n+1}$ where c is a CHR constraint.
3. Apply $\langle G, H_1 \cup H_2 \cup S, B, T\rangle_n \overset{\omega_t}{\mapsto}_P \langle C \uplus G, H_1 \cup S, \theta \wedge B, T \cup \{t\}\rangle_n$ where P contains a (renamed apart) rule $r \mathbin{@} H_1' \backslash H_2' \iff g \mid C$ and a matching substitution θ such that $chr(H_1) = \theta(H_1')$, $chr(H_2) = \theta(H_2')$, $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$, and $t = \langle id(H_1), id(H_2), r\rangle \notin T$.

Table 1. Transitions of ω_t

constraints are solved by an underlying constraint solver (e.g., the Prolog unification algorithm).

There are three types of Constraint Handling Rules: *simplification rules*, *propagation rules* and *simpagation rules*. They have the following form:

$$\begin{array}{lll} \textbf{Simplification} & r \mathbin{@} & H^r \iff g \mid B \\ \textbf{Propagation} & r \mathbin{@} H^k & \implies g \mid B \\ \textbf{Simpagation} & r \mathbin{@} H^k \setminus H^r & \iff g \mid B \end{array}$$

where r is the rule *name*, H^k and H^r are non-empty sequences of CHR constraints and are called the *heads* of the rule, the rule *guard* g is a sequence of built-in constraints, and the rule *body* B is a sequence of both CHR and built-in constraints. A program P is a set of CHR rules.

Let $vars(A)$ be the variables occurring in A, let $\bar{\exists}_A F$ denote $\exists X_1, \ldots, \exists X_n F$ where $\{X_1, \ldots, X_n\} = vars(F) \backslash vars(A)$ and let $\forall_A F$ denote $\forall Y_1, \ldots, \forall Y_n F$ where $\{Y_1, \ldots, Y_n\} = vars(A)$. Let $H = \langle H^k, H^r, g\rangle$.[1] A CHR program P forms a constraint theory consisting of the constraint theory of the built-in constraint solver in conjunction with one of the following formulas for each rule r in P (depending on its type):

$$\begin{array}{ll} \textbf{Simplification} & \forall_H((\quad g) \rightarrow (H^r \leftrightarrow \bar{\exists}_H B)) \\ \textbf{Propagation} & \forall_H((H^k \wedge g) \rightarrow (\quad \bar{\exists}_H B)) \\ \textbf{Simpagation} & \forall_H((H^k \wedge g) \rightarrow (H^r \leftrightarrow \bar{\exists}_H B)) \end{array}$$

Operationally, CHR constraints have a multi-set semantics. To distinguish between different occurrences of syntactically equal constraints, CHR constraints are extended with a unique identifier. An identified CHR constraint is denoted by $c\#i$ with c a CHR constraint and i the identifier. We write $chr(c\#i) = c$ and $id(c\#i) = i$. An alternative declarative semantics for CHR based on intuitionistic linear logic, that amongst others takes into account this multi-set semantics, is given in [3].

2.2 The Theoretical Operational Semantics

The theoretical operational semantics of CHR, denoted ω_t, is given in [10] as a state transition system. A CHR execution state σ is represented as a tuple $\langle G, S, B, T\rangle_n$ where G is the goal, a multi-set of constraints that need to be solved; S is the CHR constraint store, a set of identified CHR constraints; B is the built-in constraint store, a conjunction of built-in constraints; T is the propagation history, a set of tuples denoting the rule instances that have already fired; and n is the next free identifier, used to identify new CHR constraints. The transitions of ω_t are shown in Table 1.

The **Solve** transition solves a built-in constraint from the goal, the **Introduce** transition inserts a new CHR constraint from the goal into the CHR constraint store, and the **Apply** transition fires a rule instance. A rule instance $\theta(r)$ instantiates a rule with CHR constraints matching the heads, using matching substitution θ.

[1] We assume $H^k = \emptyset$ ($H^r = \emptyset$) for simplification (propagation) rules.

2.3 The Refined Operational Semantics

The refined operational semantics of CHR, denoted by ω_r, is introduced in [10] as a formalization of the execution mechanism of current CHR implementations. While its initial purpose was descriptive, it has become a normative description of how CHR implementations should work. This is because many CHR programs are written with the ω_r semantics in mind and often they do not work correctly under the ω_t semantics (i.e., they are not confluent). In Section 4, we present a more high-level alternative to the ω_r semantics that still allows considerable execution control.

The ω_r semantics is based on the concept of an *active* constraint. The active constraint is a CHR constraint that is used as a starting point for finding fireable rule instances. To ensure that all fireable rule instances are eventually tried, all new CHR constraints become active after they are asserted. CHR constraints that have been active before and whose variables are affected by a new built-in constraint, are reactivated.

The active constraint tries rules in textual order until either it finds a fireable rule instance or all rules have been tried. When a rule instance fires, its body is processed from left to right. Every new CHR constraint that is processed, is activated as soon as it is inserted into the constraint store. Every new built-in constraint is solved for, and all affected CHR constraints are activated one by one before processing the next constraint in the body. If the active constraint has not been removed, then after processing the rule body, it searches for then next fireable rule instance. Otherwise, processing resumes where it left before the constraint was activated.

3. Motivation and Examples

In this section, we show the benefits of our proposed language extension for some typical problems and illustrate them by examples. In these examples, we use CHR on top of Prolog, but the problems apply to other host languages as well.

3.1 Constraint Propagators

Constraint solvers generally make use of constraint propagators which filter out inconsistent values from the constraint variables' domains. Efficient solvers use a priority system to make sure that constraint propagators that are computationally cheaper or are expected to have a greater impact, are scheduled early [24, 30]. CHR rules are often used as templates for constraint propagators which are instantiated by actual constraints.

Example 1. We represent a binary constraint C between two variables X and Y as `c(`C`,`X`,`Y`)`. The domain DX of a variable X is represented as `d(`X`,`DX`)`. The following rules implement two constraint propagators:

```
ac1 @ c(C,X,Y), d(Y,DY) \ d(X,DX0) <=>
        filter(DX0,[C],[Y-DY],DX1), DX0 \= DX1 |
        d(X,DX1) pragma priority(1).
ac2 @ c(C,X,Y), d(X,DX) \ d(Y,DY0) <=>
        filter(DY0,[C],[X-DX],DY1), DY0 \= DY1 |
        d(Y,DY1) pragma priority(1).

pc1 @ c(C1,X,Y), c(C2,Y,Z), c(C3,X,Z),
    d(Y,DY), d(Z,DZ) \ d(X,DX0) <=>
        filter(DX0,[C1,C2,C3],[Y-DY,Z-DZ],DX1),
        DX0 \= DX1 | d(X,DX1) pragma priority(2).
pc2 @ c(C1,X,Y), c(C2,Y,Z), c(C3,X,Z),
    d(X,DX), d(Z,DZ) \ d(Y,DY0) <=>
```

```
        filter(DY0,[C1,C2,C3],[X-DX,Z-DZ],DY1),
        DY0 \= DY1 | d(Y,DY1) pragma priority(2).
pc3 @ c(C1,X,Y), c(C2,Y,Z), c(C3,X,Z),
    d(X,DX), d(Y,DY) \ d(Z,DZ0) <=>
        filter(DZ0,[C1,C2,C3],[X-DX,Y-DY],DZ1),
        DZ0 \= DZ1 | d(Z,DZ1) pragma priority(2).
```

In this example, the rules $\texttt{ac}_1$ and $\texttt{ac}_2$ implement arc consistency, and rules $\texttt{pc}_1$, $\texttt{pc}_2$ and $\texttt{pc}_3$ implement path consistency. Because the latter is more costly, we assign it a lower priority. We use the *pragma*[2] `priority`/1 to indicate the rule priorities. Smaller numbers denote higher priorities.

The consistencies are implemented by using the `filter`/4 predicate which filters out the inconsistent values in the domain of one of the constraint variables, given the constraints in which it appears and the domains of the other variables involved.

At first sight it might appear that, given the textual order of the rules, the refined operational semantics of CHR ensures that the arc consistency rules are always tried before the path consistency rules. The following situation shows that this is not always the case. Let `X` be a variable whose domain $\texttt{DX}_0$ has changed to $\texttt{DX}_1$ by using one of the arc consistency rules with as active constraint the domain of some variable `Y`. Assume that `X` is arc consistent. The constraint $\texttt{d(X,DX}_1\texttt{)}$ becomes active and the rules $\texttt{ac}_1$ and $\texttt{ac}_2$ are tried, none of which fires. At that moment, rule $\texttt{pc}_1$ will be tried, while there might still be a variable (e.g., `Y`) that is not arc consistent yet. In contrast, rule priorities ensure that these variables are made arc consistent first. □

3.2 Constraint Store Invariants

It is often desirable to impose certain representational invariants on the constraints in the CHR constraint store. An example of such an invariant is set semantics: no two syntactically equal constraints can exist in the constraint store. These invariants may be violated when asserting new constraints (both built-in and CHR) and we can use special purpose CHR rules for restoring them. Such rules should fire before any rule that expects (a subset of) the invariants.

Example 2. Consider that we want to check whether two graphs, G_1 and G_2 are equal. We do this by removing those edges that are common to both graphs. If there are still edges after reaching a fixed point, then the graphs are different. We represent the edges of graph G_1 and G_2 by the edge constraints $\texttt{e}_1$/2 and $\texttt{e}_2$/2 respectively. This gives us the following program:

```
s1 @ e1(X,Y) \ e1(X,Y) <=> true pragma priority(1).
s2 @ e2(X,Y) \ e2(X,Y) <=> true pragma priority(1).

rc @ e1(X,Y),  e2(X,Y) <=> true pragma priority(2).
```

Edges obey set semantics, as implemented by the rules $\texttt{s}_1$ and $\texttt{s}_2$. The rule `rc` (**r**emove **c**ommon) removes those edges that appear both in graph G_1 and in graph G_2 Now consider the query:

```
?- e1(X,X), e2(X,Y), e2(Y,X), X = Y.
```

When executing this query using the refined operational semantics, ignoring the rule priorities, the `X = Y` built-in constraint causes the sequential activation of all three CHR constraints. If the $\texttt{e}_1$/2 constraint is activated before any of the $\texttt{e}_2$/2 constraints, then rule `rc` fires before the $\texttt{s}_2$ rule is tried, which results in a final constraint store containing the $\texttt{e}_2\texttt{(X,X)}$ constraint which erroneously indicates that the graphs are different.

We already mentioned in the introduction that rule priorities require that the highest priority rule for which a fireable rule instance exists, fires. This is a global notion in that it does not matter which constraints participate in the firing rule instance, and in particular, there is no concept of an active constraint. So using rule priorities, the set semantics rules will always be tried before the lower priority rule `rc` and when the highest priority fireable rule instance is one of priority 2 (or less), then there will be no two syntactically equal $\texttt{e}_1$/2 or $\texttt{e}_2$/2 constraints.

The above example *can* be implemented correctly using the refined semantics, but this leads to inefficient and unreadable code:

```
s1 @           e1(X,Y) \ e1(X,Y) <=> true.
s2 @           e2(X,Y) \ e2(X,Y) <=> true.
s3 @ e1(_,_), e1(X,Y) \ e1(X,Y) <=> true.
s4 @ e1(_,_), e2(X,Y) \ e2(X,Y) <=> true.
s5 @ e2(_,_), e1(X,Y) \ e1(X,Y) <=> true.
s6 @ e2(_,_), e2(X,Y) \ e2(X,Y) <=> true.

rc @ e1(X,Y), e2(X,Y) <=> true.
```

□

3.3 Dynamic Rule Priorities

Rule priorities are called *dynamic* if they depend on (the arguments of) the constraints that form a rule instance. Dynamic rule priorities are only known at runtime and different instances of the same rule may have a different priority.

Example 3 (Dijkstra's Shortest Path). The program below implements Dijkstra's shortest path algorithm using dynamic rule priorities.

```
d1 @ source(V) ==> dist(V,0) pragma priority(1).
d2 @ dist(V,D1) \ dist(V,D2) <=>
              D1 < D2 | true pragma priority(1).
d3 @ dist(V,D), e(V,C,U) ==>
              dist(U,D+C)  pragma priority(D+2).
```

Here, an $\texttt{e}(V,C,U)$ constraint represents an edge from node V to node U with cost C. A `source`/1 constraint fixes the source for the (single-source) shortest path algorithm. The algorithm computes `dist`/2 constraints representing the distance of the shortest path from the source node to a given node. Rule $\texttt{d}_1$ states that the distance from the source node to itself is zero. Rule $\texttt{d}_2$ removes redundant `dist`/2 constraints. Finally, rule $\texttt{d}_3$ "labels" the nodes connected to the node that is closest to the source node and for which this has not been done yet. The (dynamic) rule priority ensures that there exists no node closer to the source node that has not been considered for labeling. □

Example 4 (Sudoku). The Sudoku solver from the CHR website [27] keeps track of the number of possible values for each Sudoku cell and chooses a value from the most constrained cell first. To do so, the following code is used:

```
fillone(N), f(A,B,C,D,N,L) <=>
          member(V,L), f(A,B,C,D,V), fillone(1).
fillone(N) <=> N < 9 | fillone(N+1).
fillone(_) <=> true.
```

The `f`/6 constraints represent a Sudoku cell: the first 4 arguments denote the position of the cell (which 3×3 block and which cell in this block); the 5^{th} argument is the number of remaining possible values for the cell; the 6^{th} argument is a list of these values. If a cell has only one possible value, it is represented by a `f`/5 constraint where the first 4 arguments again denote the position of the cell and the 5^{th} argument is the value of the cell.

Initially, the store contains the constraint `fillone(1)`. The argument of this constraint is increased until a match is found and a rule fires. After the rule has fired, it is reset to 1. In CHR^{rp} we can get the same result using only one rule:

[2] Pragmas are compiler directives. They instruct the compiler on *how* to compile a program.

1.Solve $\langle \{c\} \uplus G, S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle G, S, c \wedge B, T\rangle_n$ where c is a built-in constraint.
2. Introduce $\langle \{c\} \uplus G, S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle G, \{c\#n\} \cup S, B, T\rangle_{n+1}$ where c is a CHR constraint.
3. Apply $\langle \emptyset, H_1 \cup H_2 \cup S, B, T\rangle_n \overset{\omega_p}{\mapsto}_P \langle C, H_1 \cup S, \theta \wedge B, T \cup \{t\}\rangle_n$ where P contains a rule of priority p of the form $r \ @\ H_1' \backslash H_2' \iff g \mid C$ pragma priority(p) and a matching substitution θ such that $chr(H_1) = \theta(H_1')$, $chr(H_2) = \theta(H_2')$, $\mathcal{D} \models B \rightarrow \exists_B(\theta \wedge g)$ and $t = \langle id(H_1), id(H_2), r\rangle \notin T$. Furthermore, no rule of priority p' and substitution θ' exists with $\theta'(p') < \theta(p)$ for which the above conditions hold.

Table 2. Transitions of ω_p

```
f(A,B,C,D,N,L) <=> member(V,L), f(A,B,C,D,V)
                   pragma priority(N).
```
□

4. CHR$^{\text{rp}}$: CHR with Rule Priorities

CHR$^{\text{rp}}$ extends CHR with user-defined rule priorities. Hitherto, CHR programmers were forced to either write confluent programs that are correct under any execution strategy supported by the theoretical operational semantics of CHR, or, to take into account the low-level details of the refined operational semantics. While the latter is useful from the CHR compiler developer's perspective, it does not offer the flexible execution control requested by the CHR user. CHR$^{\text{rp}}$ forms a high-level alternative for execution control that better suits the needs of CHR programmers. In this section, we introduce the syntax and semantics of CHR$^{\text{rp}}$ and investigate its theoretical properties.

4.1 Syntax

The syntax of CHR$^{\text{rp}}$ is compatible with the syntax of (regular) CHR. A CHR$^{\text{rp}}$ simpagation rule looks as follows:

$$r \ @\ H^k \setminus H^r \iff g \mid B \text{ pragma priority}(p)$$

where r, H^k, H^r, g and B are as defined in Section 2.1. The rule *priority* p is an arithmetic expression for which holds that $vars(p) \subseteq (vars(H^k) \cup vars(H^r))$, i.e., all variables in p also appear in the heads. A rule in which $vars(p) = \emptyset$ is called a *static* priority rule: its priority is known at compile time and equal for all rule instances. A rule in which $vars(p) \neq \emptyset$ is called a *dynamic* priority rule: its priority is only known at runtime and different rule instances of the same rule may fire at different priorities.

4.2 The Priority Semantics

We propose a formal operational semantics for CHR$^{\text{rp}}$. It is called the priority semantics and denoted by ω_p. It consists of a refinement of the ω_t semantics with a minimal amount of determinism in order to support rule priorities. The ω_p semantics uses the same state representation as the ω_t semantics. Its transitions are shown in Table 2.

The ω_p semantics restricts the applicability of the **Apply** transition with respect to the ω_t semantics. It is only applicable to states with an empty goal and it fires a rule instance of priority p in state σ only if there exists no ω_t **Apply** transition $\sigma \overset{\omega_t}{\mapsto}_P \sigma'$ that fires a rule instance of a higher priority. The **Solve** and **Introduce** transitions are unchanged.

4.3 Correspondence

In this subsection, we discuss the correspondence between the ω_p semantics of CHR$^{\text{rp}}$ and the ω_t semantics of CHR. Every CHR$^{\text{rp}}$ program is a CHR program if we ignore the priority annotations. We show that every ω_p derivation is also a derivation under ω_t (Theorem 1). We then prove that the ω_p semantics respects rule priorities (Theorem 2). Finally, for CHR$^{\text{rp}}$ programs in which all rule priorities are equal, we show that every ω_t derivation corresponds to an ω_p derivation (Theorem 3).

Theorem 1. *Every derivation D under ω_p, is also a derivation under ω_t. If a state σ is a final state under ω_p, then it is also a final state under ω_t.*

Proof. The first part of the theorem holds because ω_p only adds restrictions to the applicability of ω_t transitions. For the second part, suppose that state σ is a final state under ω_p, but not under ω_t. The only transition applicable under ω_t must be the **Apply** transition, since the **Solve** and **Introduce** transitions are equal in both semantics. This means that the goal must be empty.

From all **Apply** transitions that are applicable in state σ under ω_t, we can choose the one that fires the highest priority rule instance. It is clear that this transition is also applicable under ω_p, which contradicts our assumption. This proves the second part of the theorem. □

Theorem 2. *If an **Apply** transition is applied to a state σ under ω_p, firing a rule instance of priority p, there exists no derivation under ω_t and starting in σ in which the first **Apply** transition fires a higher priority rule instance.*

Proof. The ω_p **Apply** transition, applied on state σ, fires the highest priority rule instance that can fire given the current built-in store, CHR store and propagation history. If there exists an ω_t derivation D starting in σ in which the first **Apply** transition fires a rule instance of a higher priority, then D must contain a **Solve** or **Introduce** transition that updates respectively the built-in store or CHR store, and that makes the rule instance fireable. Since the ω_p **Apply** transition requires the goal to be empty, no such derivation can exist. □

For every state $\sigma = \langle G, S, B, T\rangle_n$, there exists a derivation $\sigma \overset{\omega_p}{\mapsto}{}^*_P \sigma^*$ where $\sigma^* = \langle \emptyset, S \cup S', B \wedge B', T\rangle_{n+|S'|}$ and $G = B' \uplus chr(S')$, B' is a multi-set of built-in constraints, S' is a set of identified CHR constraints and $|S'|$ is the number of elements in S'. The derivation is formed by solving all built-in constraints, and introducing all CHR constraints in the goal G. We call state σ^* a *normalization* of σ. There are $|S'|!$ such normalizations, one for each order in which the CHR constraints of the goal are introduced.

Theorem 3. *For a given* CHR$^{\text{rp}}$ *program P in which all rule priorities are equal, it holds that for every non-failing derivation D under ω_t, if $\sigma_1 \overset{\omega_t}{\mapsto}_P \sigma_2 \in D$ then for every normalization σ_2^* of σ_2, there exists a normalization σ_1^* of σ_1 such that $\sigma_1^* \overset{\omega_p}{\mapsto}{}^*_P \sigma_2^*$. If a state σ is a final state under ω_t, it is also a final state under ω_p.*

Proof. Given $\sigma_1 \overset{\omega_t}{\mapsto}_P \sigma_2 \in D$. We look at each of the three possible transitions:

1. **Solve** $\sigma_1 = \langle \{c\} \uplus G, S, B, T\rangle_n$ and $\sigma_2 = \langle G, S, c \wedge B, T\rangle_n$. Clearly all normalizations of σ_1 and σ_2 are equal.
2. **Introduce** $\sigma_1 = \langle \{c\} \uplus G, S, B, T\rangle_n$ and $\sigma_2 = \langle G, \{c\#n\} \cup S, B, T\rangle_{n+1}$. All normalizations of σ_2 are also normalizations of σ_1.
3. **Apply** $\sigma_1 = \langle G, H^r \cup S, B, T\rangle_n$ and $\sigma_2 = \langle C \uplus G, S, \theta \wedge B, T'\rangle_n$. In any normalization of σ_1, the same rule instance can fire because introducing CHR constraints to the store nor

solving built-in constraints from the goal can prevent a rule instance from being fireable.[3] So we have for every normalization σ_1^* of σ_1 that $\sigma_1^* = \langle \emptyset, H^r \cup S \cup S', B \wedge B', T \rangle_{n'} \overset{\omega_p}{\longmapsto}_P \langle C, S \cup S', \theta \wedge B \wedge B', T' \rangle_{n'} = \sigma_2'$. It is easy to see that every normalization of σ_2 corresponds to a normalization of such a state σ_2' for some normalization of σ_1.

We conclude that for every transition $\sigma_1 \overset{\omega_t}{\longmapsto}_P \sigma_2$, there exists a corresponding derivation $\sigma_1^* \overset{\omega_p}{\longmapsto}_P \sigma_2^*$ for every normalization σ_2^* of σ_2. The second part of the theorem follows from Theorem 1. □

Theorem 3 implies that for CHR$^{\mathrm{rp}}$ programs in which all rule priorities are equal, every execution strategy under ω_t is consistent with ω_p, and so such programs can be executed using the refined operational semantics as implemented by current CHR implementations. While such CHR$^{\mathrm{rp}}$ programs are obviously degenerate, execution under the ω_r semantics or slightly altered versions of it, is possible for a larger class of programs. This is illustrated in Section 5, where we show how CHR$^{\mathrm{rp}}$ programs with static rule priorities can be executed in CHR implementations based on the ω_r semantics, by using a straightforward source-to-source transformation.

4.4 Confluence

In [1], a criterion is given for deciding whether a (terminating) CHR program is confluent under the ω_t semantics. Confluence is based on the concept of joinability:

Definition 1 (Joinability). *Two states σ_1 and σ_2 are joinable given program P and operational semantics ω if $\sigma_1 \overset{\omega}{\longmapsto}_P^* \sigma_1^*$ and $\sigma_2 \overset{\omega}{\longmapsto}_P^* \sigma_2^*$ with σ_1^* and σ_2^* variants with respect to the common variables of σ_1 and σ_2.*

Two execution states are variants with respect to a set of variables $\mathcal{V}$ if they are identical modulo renaming of constraint identifiers, and after projection of the built-in constraints on the variables in $\mathcal{V}$ and removal of propagation history tuples involving removed CHR constraints.

Definition 2 (Local Confluence). *A CHR program P is locally confluent under operational semantics ω if for all states σ, σ_1 and σ_2: if $\sigma \overset{\omega}{\longmapsto}_P \sigma_1$ and $\sigma \overset{\omega}{\longmapsto}_P \sigma_2$ then σ_1 and σ_2 are joinable.*

If P terminates then local confluence implies confluence.

Definition 3 (Confluence). *A CHR program P is confluent under operational semantics ω if for all states σ, σ_1 and σ_2: if $\sigma \overset{\omega}{\longmapsto}_P^* \sigma_1$ and $\sigma \overset{\omega}{\longmapsto}_P^* \sigma_2$ then σ_1 and σ_2 are joinable.*

Under ω_t, local confluence can be proven by checking all *critical pairs* for joinability. A critical pair consists of two *minimal* states σ_1 and σ_2 which follow from firing respectively rule instance $\theta_1(r_1)$ and $\theta_2(r_2)$ in common ancestor state σ such that $\theta_2(r_2)$ cannot fire in σ_1 and $\theta_1(r_1)$ cannot fire in σ_2. Clearly, under ω_p, a critical pair only follows from firing rule instances with equal priority.

If a rule instance $\theta(r)$ is fireable in state $\langle G, S, B, T \rangle_n$ under ω_t, then it this is also the case in any (non-failed) "larger" state $\langle G \uplus G', S \cup S', B \cup B', T \setminus T' \rangle_{n'}$. This result does not hold under ω_p because adding CHR or built-in constraints to the store or removing propagation history tuples, can cause a *higher priority* rule instance to become fireable. A similar problem was found in [9] for the refined operational semantics, although its exact cause was not clearly identified.

[3] We do not consider non-monotone guards like Prolog's `var`/1. They are not allowed in pure CHR.

Example 5. Consider the following example, adapted from [9] and extended with rule priorities:

```
r1 @ p, q(_) <=> r pragma priority(1).
r2 @ q(_), q(_) \ r <=> true pragma priority(2).
r3 @ r \ q(_) <=> true pragma priority(3).
r4 @ r <=> true pragma priority(4).
```

A critical pair for rule `r1` is

$$(\langle \{r\}, \{q(1)\#1\}, true, T_1 \rangle_n, \langle \{r\}, \{q(2)\#2\}, true, T_2 \rangle_n)$$

with common ancestor state

$$\langle \emptyset, \{q(1)\#1, q(2)\#2, p\#3\}, true, T \rangle_n$$

Both states further derive into $\langle \emptyset, \emptyset, true, T' \rangle_{n+1}$ and so it seems that there is confluence. However, given the initial goal $\{p, q(1), q(2), q(3)\}$ we get the final stores $\{q(1), q(2)\}$, $\{q(2), q(3)\}$ or $\{q(1), q(3)\}$. The problem is that in a minimal state, rule `r2` is not applicable and rules `r3` and `r4` can fire. In a larger state, `r2` may become applicable and cause rules `r3` and `r4` to be not applicable anymore. □

Clearly, it is not sufficient to look at minimal states only. We can limit the number of potential states by noting that in the ancestor state of a critical pair, no higher priority rule instance could fire. A practical confluence test is however outside the scope of this paper.

4.5 Examples

We illustrate the different behavior of the ω_p semantics and the ω_r semantics on some small examples. The first example shows that rule priorities are "stronger" than rule order under the ω_r semantics.

Example 6. Consider the following program:

```
r1 @ a ==> write('rule 1\n'), b pragma priority(1).
r2 @ a, b ==> write('rule 2\n') pragma priority(2).
r3 @ a    <=> write('rule 3\n') pragma priority(3).
r4 @ a, b ==> write('rule 4\n') pragma priority(4).
```

Using the refined operational semantics, the rule priority declarations are ignored. For the initial goal `a`, we get the following output:

ω_r Semantics	ω_p Semantics
rule 1	rule 1
rule 2	rule 2
rule 4	rule 3
rule 3	

In the ω_p semantics, constraint `a` is removed by rule `r3` before rule `r4` is tried. This causes rule `r4` to be not applicable anymore. There is no rule ordering that can cause rule `r3` to be fired after rule `r2` but before rule `r4` in the ω_r semantics. □

The following two examples illustrate the non-determinism in the ω_p semantics. Note that this non-determinism could be removed by further refining the ω_p semantics (e.g., using rule order, recency, etc. to resolve conflicts). However, we prefer to keep all the determinism users can rely on, explicit in the priority annotations.

Example 7. Consider the following program:

```
r1 @ a(X) ==> write(r1:X), nl pragma priority(1).
r2 @ a(X) ==> write(r2:X), nl pragma priority(2).
```

For the initial goal `a(1), a(2)`, we get the following output:

ω_r Semantics	ω_p Semantics			
r1:1	r1:1	r1:1	r1:2	r1:2
r2:1	r1:2	r1:2	r1:1	r1:1
r1:2	r2:1	r2:2	r2:1	r2:2
r2:2	r2:2	r2:1	r2:2	r2:1

Here the non-determinism is caused by the existence of different rule instances for the same rule. □

Example 8. Consider the following program:

```
r1 @ a ==> write('rule 1\n') pragma priority(1).
r2 @ a ==> write('rule 2\n') pragma priority(1).
```

In this example, rules r1 and r2 have an equal priority. For the initial goal a, we get the following output:

ω_r Semantics	ω_p Semantics	
rule 1	rule 1	rule 2
rule 2	rule 2	rule 1

Here, the non-determinism is caused by the existence of different rules with equal priority. □

A final example compares the ω_p semantics with the ω_t semantics and shows that CHR$^{\mathrm{rp}}$ programs are not always monotonic.

Example 9. A form of negation by absence can be implemented as follows:

```
r1 @ a \ no_a <=> fail pragma priority(1).
r2 @     no_a <=> true pragma priority(2).
```

Under the ω_t semantics, the goal `a, no_a` either fails or succeeds, whereas under the ω_p semantics it must fail. The goal a succeeds under both semantics. □

5. Translating CHR$^{\mathrm{rp}}$ into CHR

In this section, we present a source-to-source transformation that transforms a CHR$^{\mathrm{rp}}$ program with *static* rule priorities into a CHR program that is equivalent when executed under the ω_r semantics.[4] We make use of Prolog as host language, but the transformation can easily be adapted to work with other host languages. We also make use of some compiler directives of the K.U.Leuven CHR system [26] for reasons of efficiency and conciseness.[5]

The implementation proposed in this section depends on some details of the ω_r semantics. Therefore, we briefly review its state representation and transitions. More details can be found in [10]. An ω_r execution state is a tuple $\langle A, S, B, T\rangle_n$ with S, B, T and n as in the ω_t semantics and A the execution stack: a sequence of built-in constraints and ((occurrenced) identified) CHR constraints. An occurrenced identified CHR constraint is represented as $c\#i : j$ and denotes the identified CHR constraint $c\#i$ which is considered as active constraint in its j^{th} occurrence in the program P. The transitions of ω_r are shown in Table 3.

The proposed translation implements the following 4 changes to the refined operational semantics to make it compatible with the priority semantics:

1. A new CHR constraint is not activated when it is introduced to the store by the **Activate** transition. Instead, it is scheduled for activation at each priority for which it has an occurrence.
2. A CHR constraint that is reconsidered by the **Solve** transition, is not reactivated by the **Reactivate** transition when it appears on top of the execution stack. Instead, it is rescheduled for activation at each priority for which it has an occurrence.
3. A constraint is activated at a given priority: the *current* priority. It only tries rules of this priority.
4. After processing the initial goal (query) and at the end of processing a rule body, the highest priority scheduled constraint becomes active if its priority is higher than the current priority. Otherwise, the current active constraint remains active.

[4] Some of the non-determinism introduced by the ω_p semantics is however lost by the translation.

[5] These are the *pragmas* passive/1, history/2 and no_history/0. Their semantics is explained further on.

1. Solve $\langle [c \mid A], S_0 \cup S_1, B, T\rangle_n \overset{\omega_r}{\mapsto}_P \langle S_1 +\!\!+ A, S_0 \cup S_1, c \wedge B, T\rangle_n$ where c is a built-in constraint and $vars(S_0) \subseteq$ *fixed*(B) where *fixed*(B) are the variables fixed by B.
2. Activate $\langle [c \mid A], S, B, T\rangle_n \overset{\omega_r}{\mapsto}_P \langle [c\#n : 1 \mid A], \{c\#n\} \cup S, B, T\rangle_{n+1}$ where c is a CHR constraint.
3. Reactivate $\langle [c\#i \mid A], S, B, T\rangle_n \overset{\omega_r}{\mapsto}_P \langle [c\#i : 1 \mid A], S, B, T\rangle_n$ where c is a CHR constraint.
4. Drop $\langle [c\#i : j \mid A], S, B, T\rangle_n \overset{\omega_r}{\mapsto}_P \langle A, S, B, T\rangle_n$ if CHR constraint c has no j^{th} occurrence in P.
5. Simplify $\langle [c\#i : j \mid A], \{c\#i\} \cup H_1 \cup H_2 \cup H_3 \cup S, B, T\rangle_n \overset{\omega_r}{\mapsto}_P \langle C +\!\!+ A, H_1 \cup S, \theta \wedge B, T\rangle_n$ where the j^{th} occurrence of c in a (renamed apart) rule in P is $r \mathbin{@} H_1' \backslash H_2', d_j, H_3' \iff g \mid C$ and there exists a matching substitution θ such that $c = \theta(d_j)$, $chr(H_1) = \theta(H_1')$, $chr(H_2) = \theta(H_2')$, $chr(H_3) = \theta(H_3')$ and $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$.
6. Propagate $\langle [c\#i : j \mid A], H^k \cup H_3 \cup S, B, T\rangle_n \overset{\omega_r}{\mapsto}_P \langle C +\!\!+ [c\#i : j \mid A], H^k \cup S, \theta \wedge B, T \cup \{t\}\rangle_n$ where $H^k = \{c\#i\} \cup H_1 \cup H_2$ and the j^{th} occurrence of c in a rule in P is $r \mathbin{@} H_1', d_j, H_2' \backslash H_3' \iff g \mid C$ and there exists a matching substitution θ such that $c = \theta(d_j)$, $chr(H_1) = \theta(H_1')$, $chr(H_2) = \theta(H_2')$, $chr(H_3) = \theta(H_3')$, $\mathcal{D} \models B \rightarrow \bar{\exists}_B(\theta \wedge g)$, and $t = id(H_1) +\!\!+ [i] +\!\!+ id(H_2) +\!\!+ id(H_3) +\!\!+ [r] \notin T$.
7. Default $\langle [c\#i : j \mid A], S, B, T\rangle_n \overset{\omega_r}{\mapsto}_P \langle [c\#i : j+1 \mid A], S, B, T\rangle_n$ if no other transition applies.

Table 3. Transitions of ω_r

We now show how each of these changes can be implemented as a source-to-source transformation.

5.1 Extended Constraint Representation

For every CHR constraint $c(\bar{X})$ of the original program, an extended CHR constraint $c'(\bar{X}, T)$ is made where T is a fresh Prolog variable. T uniquely identifies a constraint and is used to trigger (reactivate) this constraint on demand. We call it the *trigger variable*. The rules generating the extended constraint representations have the following form:

$c(\bar{X})$ `<=>` $c'(\bar{X}$`,_)`.

5.2 Representation of the Current Priority

The current priority is represented as a priority/1 constraint. At all times, the CHR constraint store contains exactly one such constraint. In the initial state, the constraint store contains the constraint priority($P_\perp$) with $P_\perp$ lower than any rule priority in the program P. The current priority is changed by asserting a set_priority/1 constraint which fires the following rule:

```
set_priority(P), priority(_) <=> priority(P).
```

5.3 Scheduling Constraints

A constraint $c(\bar{X})$ (with extended representation $c'(\bar{X}, T)$) is scheduled by the following rule:

```
priority(P⊥)#Id, c'(X̄,T) ==> trigger(T) |
        insert(P1,T), ..., insert(Pn,T)
        pragma passive(Id), no_history.
```

This rule can only fire if there is a priority($P_\perp$) constraint in the store. This allows us to distinguish between "normal" activation (after introducing the CHR constraint) and reactivation (after asserting a built-in constraint that affects the CHR constraint's arguments) on the one hand, and reactivation "on demand" on the other.

The pragma passive/1 states that the constraint referred to by its argument (the priority/1 constraint in this case) does not try this rule, i.e., when a new priority/1 constraint is asserted, the rule is not considered. The pragma no_history/0 states that no propagation history is to be used for this rule, i.e., it can fire multiple times using the same combination of CHR constraints.

The trigger(T) guard ensures that built-in constraints on trigger variable T will reactivate the constraint $c'(\bar{X},T)$ (see Section 5.5). The trigger/1 predicate equals true.

5.4 Rule Transformation

Every CHR$^{\text{rp}}$ rule

r @ $c_1(\bar{X}_1)$, ..., $c_{i-1}(\bar{X}_{i-1})$ \ $c_i(\bar{X}_i)$, ..., $c_n(\bar{X}_n)$
<=> *guard* | *body* pragma priority(P).

is transformed into a CHR rule

r @ priority(P)#Id_0,
$c'_1(\bar{X}_1,\_)$#Id_1, ..., $c'_{i-1}(\bar{X}_{i-1},\_)$#$Id_{i-1}$ \
$c'_i(\bar{X}_i,\_)$#Id_i, ..., $c'_n(\bar{X}_n,\_)$#Id_n <=> *guard* |
set_priority($P_\perp$), *body*, set_priority(P),
check_activation pragma passive(Id_0),
history(r,[Id_1,...,Id_n]).

The rule only fires if a priority(P) constraint is in the store, i.e., if the current priority equals the rule priority. In the body, the current priority is first changed into $P_\perp$. Then the original body is executed. All new CHR constraints, as well as those affected by built-in constraints, are scheduled when they are (re-)activated using the rules introduced in the Section 5.3. After the body has been processed, the current priority is changed back to P and it is checked whether a higher priority rule should be tried using check_activation/0 (see the Section 5.6). The current priority does not need to be changed if the body does not contain CHR constraints nor built-in constraints that might affect CHR constraints already in the store. This is for example the case if the body equals true. If the current priority already is the highest priority, it is not needed to call check_activation/0.

The pragma history(r,ids) states that the propagation history used by this rule consists of sequences formed by appending name r to the identifiers of the constraints referred to by ids. In this case, the propagation history differs from the default history in that the identifier of the priority/1 constraint is not included. If the original rule r is a simplification or simpagation rule, the history/2 annotation is not needed.

5.5 Reactivation On Demand

On demand reactivation is somewhat tricky. Whenever a built-in constraint c is solved, the **Solve** transition reactivates at most all non-ground CHR constraints and at least one constraint from each rule instance that has become fireable by solving c. Given the lower bound, it is not possible to reactivate a given CHR constraint on demand an unbounded number of times using a finite number of rules. This is because each reactivation fires at least one rule instance in which the reactivated constraint participates. For single-headed rules, this requires one rule for each reactivation. For multi-headed rules, it is not guaranteed which constraint is reactivated.

On the other hand, every CHR implementation based on the refined operational semantics implements the **Reactivate** transition. In the K.U.Leuven CHR system, we use

```
reactivate(T) :- ( get_attr(T,Mod,Attr) ->
                   Mod:attr_unify_hook(Attr,_) ; true ).
```

with *Mod* the name of the module of the CHR program.[6] This simulates a **Solve** transition in which the referenced constraint is reactivated. In the K.U.Leuven JCHR system [32], on demand reactivation can be implemented using user-defined built-in constraints.

5.6 Checking the Schedule

The next two rules check whether the current priority should change after firing a rule. The first rule fires if the highest priority scheduled constraint has a priority higher than the current priority. If so, it is removed from the schedule, the current priority is changed, the constraint is activated, and then the priority is changed back to the previous current priority after which the process is repeated. The second rule fires if the current priority is higher or equal to the priority of the highest priority scheduled constraint (if any). In that case, the current active constraint remains active.

```
priority(P) \ check_activation <=>
        find_min(MP,T), MP < P | delete_min(MP),
        set_priority(MP), reactivate(T),
        set_priority(P), check_activation.
check_activation <=> true.
```

Example 10. Consider the following program with initial goal {a}:

```
r1 @ a ==> b,c  pragma priority(4).
r2 @ b <=> d    pragma priority(2).
r3 @ c <=> true pragma priority(3).
r4 @ d <=> true pragma priority(1).
r5 @ a <=> true pragma priority(4).
```

Figure 1 shows the resulting derivation. After processing the initial goal, the first call to check_activation/0 (*activation call* henceforth) activates constraint a at priority 4, which fires rule r_1. This rule schedules constraints b and c. The activation call after firing r_1 wakes up constraint b (the current priority is then 2) which fires rule r_2. Constraint d is scheduled and activated after the activation call following the firing of r_2. It fires rule r_4. Because of its trivial body (and its priority), no activation call is needed after processing the rule body. The activation call following the activation of d silently succeeds. The one following the activation of b wakes up c which fires rule r_3. Again, because of the trivial rule body, no activation call is needed after firing rule r_3. The call following the activation of c also silently succeeds after which a becomes active again. Constraint a searches for the next fireable rule instance which is an instance of rule r_5. Finally, the activation call following the activation of a silently succeeds and the execution stops. □

If after firing a rule, a higher priority constraint needs to be reactivated, the execution stack stores the context of the current active constraint so that it can proceed where it left after the reactivation.

5.7 The Initial Goal

Every initial CHR$^{\text{rp}}$ goal G is transformed into

```
?- priority(P⊥), G, check_activation.
```

First, the priority is set to $P_\perp$. Then, the goal is processed and its CHR constraints are scheduled. Finally, the highest priority scheduled constraint is activated.

[6] The trigger variable has an attribute containing the constraints in which it is contained. If such an attribute does not exist, this means that the constraint has already been removed.

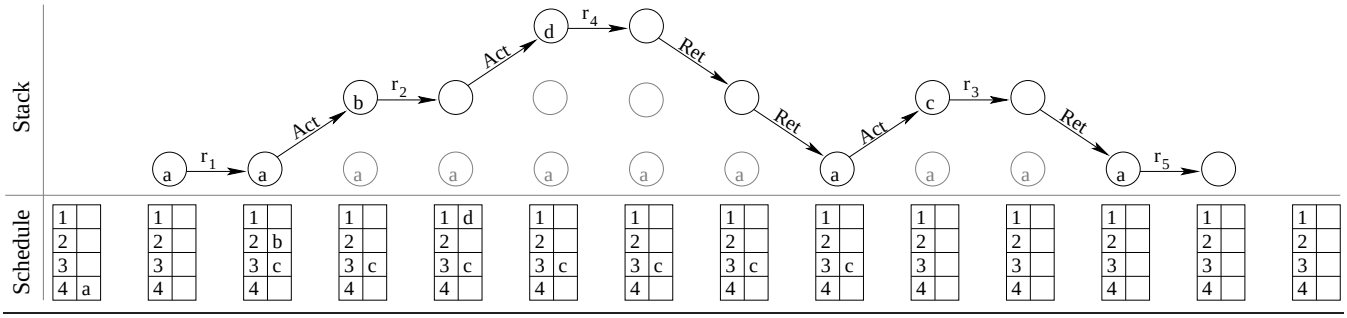

Figure 1. Activation Example

5.8 The Priority Queue

The priority queue module defines the `insert`/2, `find_min`/2 and `delete_min`/1 predicates.[7] These predicates are implemented in CHR by the following rules:

```
fm @ find_min(QPrio,QItem) <=> fm(1,QPrio,QItem).

fm_found @ insert(Prio,Item) \ fm(Prio,QPrio,QItem)
        <=> QPrio = Prio, QItem = Item.
fm_next  @ insert(_,_) \ fm(Prio,QPrio,QItem) <=>
        fm(Prio + 1,QPrio,QItem).
fm_fail  @ fm(_,_,_) <=> fail.

dm_found @ insert(P,_), delete_min(P) <=> true.
dm_fail  @ delete_min(_) <=> fail.
```

The `insert`/2 constraints are merely inserted into the constraint store. Rule `fm` transforms a `find_min`/2 constraint into a `fm`/3 constraint which has a *current* priority as first argument. This initiates the search for the highest priority item by looking for items with priority 1. If an item is found with priority equal to the current priority, this is the highest priority item and it is returned using rule `fm_found`. Otherwise, given that there are still items in the queue, rule `fm_next` decreases the current priority. If there are no items left in the queue, rule `fm_fail` causes failure. Rule `dm_found` deletes the first item with priority equal to the argument of a `delete_min`/1 constraint. Rule `dm_fail` fails when no such item exists.

In this implementation, inserting and deleting an item take constant time. Given that all priorities are between 1 and N, finding the highest priority item takes $\mathcal{O}(N)$ time. For any given program, N is constant.

5.9 Example

In this section, we show the complete transformation of an example program. The program has no intended meaning, but illustrates some of the variations in the translation of rules that depend on the type of the rule (simplification, simpagation or propagation), its body and its priority. The CHR$^{\mathrm{rp}}$ program

```
r1 @ a(X), b(X) ==> b(X) pragma priority(1).
r2 @ b(X) \ b(X) <=> true pragma priority(2).
r3 @ a(X) <=> X > 0 | b(X) pragma priority(3).
```

is translated into

```
a(X) <=> a(X,_).
b(X) <=> b(X,_).

priority(4) #Id0, a(_,T) ==>
        trigger(T) | insert(1,T), insert(3,T)
        pragma passive(Id0), no_history.
priority(4) #Id0, b(_,T) ==>
        trigger(T) | insert(1,T), insert(2,T)
        pragma passive(Id0), no_history.

trigger(_).

r1 @ priority(1) #Id0, a(X,_) #Id1, b(X,_) #Id2 ==>
        set_priority(4), b(X), set_priority(1)
        pragma passive(Id0), history(r1,[Id1,Id2]).
r2 @ priority(2) #Id0, b(X,_) \ b(X,_) <=>
        true pragma passive(Id0).
r3 @ priority(3) #Id0, a(X,_) <=> X > 0 |
        set_priority(4), b(X), set_priority(1),
        check_activation pragma passive(Id0).

set_priority(P), priority(_) <=> priority(P).
set_priority(_) <=> fail.

priority(P) \ check_activation <=>
        find_min(MP,T), MP < P | delete_min(MP),
        set_priority(MP), reactivate(T),
        set_priority(P), check_activation.
check_activation <=> true.

reactivate(T) :- ( get_attr(T,user,Attr) ->
                   attr_unify_hook(Attr,_) ; true ).
```

In the translation of rule $\mathtt{r}_1$ we do not check whether a constraint is scheduled for activation at a higher priority because the current priority already is maximal. The body of rule $\mathtt{r}_2$ does not add any built-in or CHR constraint and so we do not change the current priority before and after processing its body, nor do we check for constraints scheduled at a higher priority. Because it is a simpagation rule, no propagation history is needed. Rule $\mathtt{r}_3$ is a simplification rule and so again no propagation history is needed.

6. Correspondence and Overhead

6.1 Correspondence

Consider the following abstraction function

$$\alpha(\langle A, S, B, T\rangle_n) = \langle G, S', B, T\rangle_n$$

$$\text{with } \begin{cases} G = \{c \mid c \text{ a built-in constraint} \wedge c \in A\} \uplus \\ \qquad \{c(\bar{X}) \mid c'(\bar{X}, \_) \in A \vee c(\bar{X}) \in (A \uplus chr(S))\} \\ S' = \{c(\bar{X})\#i \mid c'(\bar{X}, \_)\#i \in A\} \end{cases}$$

[7] The `delete_min`/1 predicate has the queue's current highest priority as argument to simplify the implementation.

The following theorem shows that our translation implements the original program.[8]

Theorem 4. *Given a state σ of the translated program $T(P)$ then every derivation $\sigma \overset{\omega_r}{\mapsto}{}^*_{T(P)} \sigma'$ corresponds to a derivation $\alpha(\sigma) \overset{\omega_p}{\mapsto}{}^*_P \alpha(\sigma')$ and if σ is a final state of $T(P)$ under ω_r then $\alpha(\sigma)$ is a final state of P under ω_p.*

Proof. The ω_r semantics implements the ω_t semantics. The translated program $T(P)$ restricts the applicability of the rules of the original program P and does not change their result with respect to the abstraction function α. The extra rules introduced by the translation have no effect with respect to α. Hence every derivation $\sigma \overset{\omega_r}{\mapsto}{}^*_{T(P)} \sigma'$ corresponds to a derivation $\alpha(\sigma) \overset{\omega_t}{\mapsto}{}^*_P \alpha(\sigma')$ (ignoring the rule priorities).

Under ω_p, a rule can only fire if the goal is empty. In the translation, while processing the initial goal or a rule body, no rule corresponding to an original program rule can fire because the priority is set to $P_\perp$. When the processing is done, all built-in constraints have been solved, all CHR constraints are introduced and all constraints of the form $c(\bar{X})$ have been transformed into constraints of the form $c'(\bar{X}, T)$ and so the goal is abstracted to $\emptyset$.

When a rule instance $\theta(r)$ fires in the translation in state σ, no higher priority rule instance exists that can fire in the original program in state $\alpha(\sigma)$ under ω_p. If such a rule instance, say $\theta'(r')$, would exist, there exists an event (assertion of a CHR constraint or built-in constraint) that made the rule instance fireable. This event must have activated at least one of the CHR constraints that participate in $\theta'(r')$. This constraint is then scheduled at the priority of r'. If it is activated at this priority, then $\theta'(r')$ must have fired before trying lower priority rules, and so (because of the propagation history) it cannot fire again. This contradicts our assumption. If the scheduled constraint is not activated, then it must still be in the priority queue, and since $\theta(r)$ can only have fired after a call to `check_activation` shows that there is no higher priority scheduled constraint, this also contradicts our assumption.

Using similar reasoning it can be shown that if σ is a final state in the translation under ω_r, then $\alpha(\sigma)$ must also be a final state in the original program under ω_p. □

6.2 Overhead

In this subsection, we look at the overhead introduced by our transformation. Obviously, the execution strategy influences the complexity of a given program and so a general comparison of the complexity before and after the transformation is not feasible. Therefore, we make abstraction of the execution strategy followed and only look at the overhead caused by the different changes our transformation produces with respect to the original program. In the rest of this section, we assume that rule priorities are between 1 and N for some N which is constant for each individual program.

6.2.1 Runtime Overhead

Scheduling and Activation The transformation breaks up the **Activate** and **Reactivate** transitions in a scheduling phase and an activation phase. Each constraint is scheduled at each priority at which it has occurrences, which takes $\mathcal{O}(N)$ time. Each scheduling operation requires an insertion into the priority queue, which takes constant time. The activation requires finding the constraint that needs to be activated and causing a **Reactivate** transition. Because the trigger variables uniquely identify constraints, both these operations take constant time.

[8] Note that only CHR$^{\text{rp}}$ programs with static rule priorities can be translated using our scheme.

Trying Rules The active constraint considers all its occurrences in the program. We distinguish between occurrences in rules whose priority equals the current priority, and occurrences for which this is not the case.

The cost of trying a rule with priority equal to the current priority is equivalent to the cost of trying the original rule under the refined operational semantics. This is because the transformed rule only contains one extra head representing the current priority. Because there always is exactly one `priority`/1 constraint, it can be found in constant time and it does not increase the number of partial matches. Hence the cost of finding partner constraints for a given rule is proportional to the cost in the original program given the same active constraint and the same state (after applying the abstraction function of Section 6.1).

The cost of trying a rule with priority different from the current priority crucially depends on the *join ordering* used by the CHR implementation. The join ordering is the order in which the active constraint looks up its partner constraints. If the `priority`/1 constraint is the first partner constraint, then since in any state, there is exactly one such constraint, the cost of trying such a rule is constant. By adding a rule

```
priority(_) \ priority(_) <=> true.
```

the K.U.Leuven CHR system derives that the `priority`/1 constraint can have at most one instance and takes this fact into account when generating a join ordering. Although there are no guarantees that the `priority`/1 constraint is chosen as the first partner constraint, in practice this is most often the case. Note that in an optimized implementation of CHR$^{\text{rp}}$, rules with incorrect priority would not be considered at all, reducing the cost per rule from $\mathcal{O}(1)$ to $\mathcal{O}(0)$.

Checking the Schedule After each rule firing, the schedule is checked to see whether a constraint is scheduled for activation at a priority higher than the current one. This requires finding the highest priority scheduled constraint in the priority queue and comparing its priority with the current. If it is higher than the current priority, it is deleted from the priority queue, and the constraint is activated. Finding the highest priority item takes $\mathcal{O}(N)$ time; the other operations can be completed in constant time.

6.2.2 Space Overhead

In the transformation, every constraint is extended with a trigger variable. Furthermore, we maintain a priority queue. This priority queue contains at most N elements for each constraint. The priority queue requires space proportional to the number of elements in it. In total, the space overhead is linear in the size of the constraint store.

6.2.3 Propagation History

The K.U.Leuven CHR compiler can detect that no propagation history is necessary for a rule, for example if it has only one active constraint which can never be reactivated. This can cause a considerable reduction in space usage. By adding an explicit reactivation mechanism, the analyses conservatively apply no optimization, because the priority system obfuscates the execution strategy. If after manual analysis of the code, it is found that a history is not necessary, we can instruct the compiler not to use a history by using the pragma `no_history`.

7. Experimental Evaluation

In this section, we present the results of experiments evaluating the runtime performance of our implementation. In Section 7.1 we apply the source transformation to execute the simple arc and path consistency program presented in Section 3.1 and compare

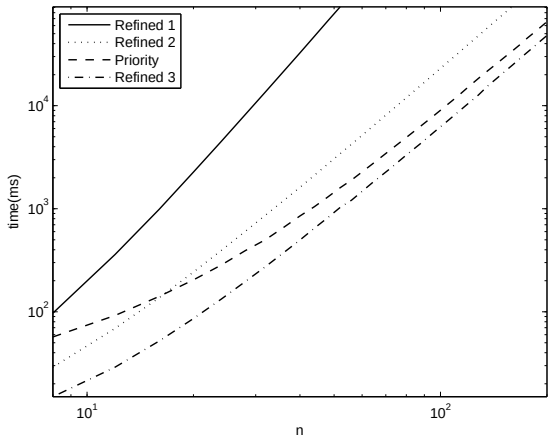

Figure 2. Constraint Propagation Runtimes

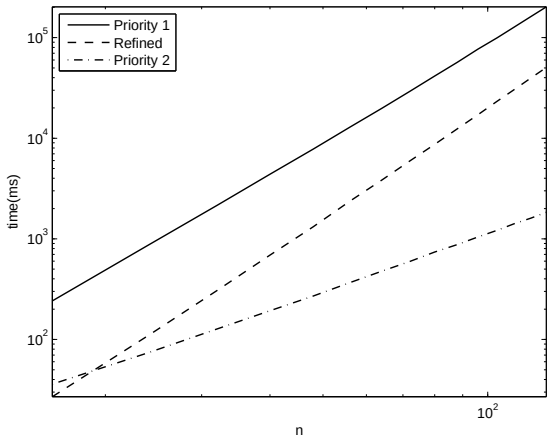

Figure 3. Less or Equal Runtimes

with execution under the refined operational semantics. In Section 7.2 we experiment with a partial order constraint solver. Finally, Section 7.3 evaluates the constant time overhead generated by the scheduling mechanism. The experiments are performed on a Pentium IV 2.8 GHz using SWI-Prolog 5.6.28. The runtimes do not include time spent on garbage collection.

7.1 Constraint Propagators

We applied the arc and path consistency program from Section 3.1 to the following constraints: $a \leq b$, $b \leq c$, $c < a$, $d \leq a$, $d \leq b$ and $d \leq c$. All variables have as initial domain the interval $[-n, n]$ for given n. The constraints between a, b and c are inconsistent and this inconsistency can be found using arc consistency alone using $\mathcal{O}(n)$ iterations. The constraints involving d have no impact. However, when the domain of d changes, the changed domain is used as active constraint under the refined operational semantics. After trying the arc consistency rules, it tries the considerably more expensive path consistency rules.

Figure 2 shows a log-log plot of runtime versus n. The results under the refined semantics are very variable. They depend on the order in which constraints appear in the goal. In a worst case scenario (Refined 1) the domain constraint of d becomes active in every iteration and the total runtime is $\mathcal{O}(n^4)$. By reordering some of the constraints in the goal, this worst case scenario is avoided (Refined 2). The best results are achieved by postponing the insertion of the domain of d until after arc consistency is enforced on a, b and c (Refined 3). In the latter two cases, the complexity is $\mathcal{O}(n^3)$. The results of using our static priority translation (Priority) are more or less independent of the order of the constraints in the goal. The scheduling overhead of our transformation is proportional to the number of rule firings which is $\mathcal{O}(n)$. It is visible in the left part of the plot of Priority.

7.2 Less or Equal

The Less or Equal benchmark uses the following program:

```
leq(X,X) <=> true pragma priority(1).
leq(X,Y), leq(Y,X) <=> X = Y pragma priority(1).
leq(X,Y) \ leq(X,Y) <=> true pragma priority(1).
leq(X,Y), leq(Y,Z) ==> leq(X,Z) pragma priority(2).
```

and for given n, the initial goal

$$G = \{\texttt{leq}(X_1,X_2), \ldots, \texttt{leq}(X_{n-1}, X_n), \texttt{leq}(X_n,X_1)\}$$

It derives into a state in which $X_1 = X_2 = \ldots = X_{n-1} = X_n$. Figure 3 shows a log-log plot of the benchmark results. We use two different setups to test our static priority implementation and compare with the results under the refined operational semantics (Refined). In a first benchmark (Priority 1), the subgoal $G \setminus \{\texttt{leq}(X_n,X_1)\}$ is processed first and in a second phase the constraint $\texttt{leq}(X_n,X_1)$ is added. The first phase behaves similar as under the refined operational semantics. However, in the second phase there is a difference.

The constraint $\texttt{leq}(X_n,X_1)$ fires the second rule (anti-symmetry) which causes X_1 to be unified with X_n. Let X be the result of this unification. At that moment, all constraints in which X appears, are activated in some undefined order. Amongst these constraints is the constraint $\texttt{leq}(X,X)$. If this constraint is not activated first, it will be used as a partner constraint in the final rule (transitivity) for each constraint $\texttt{leq}(X,X_i)$ with $2 \leq i \leq n-1$. In the static priority implementation, the constraint $\texttt{leq}(X,X)$ becomes active at priority 1 before any constraint becomes active at priority 2 and so this situation does not occur. The plot shows that although it has worse constant factors, this setup scales better than when using the refined operational semantics.

In a second test, we process the goal completely before trying rules. In this case, because the implementation prefers more recent constraints from those scheduled for activation, the priority 2 rule fires only $\mathcal{O}(n)$ times and clearly, the benchmark scales better than both Refined and Priority 1.

7.3 Constant Factors

In this subsection, we measure the overhead that our implementation introduces in the form of constant factors. This overhead consists of retrieving the current priority for each rule that is tried and checking whether it is equal to the rule priority. If this is not the case, the rule is skipped. When a rule fires, the current priority is changed before and after processing the rule body, CHR constraints introduced or awoken by the body are scheduled and it is checked whether a constraint is scheduled at a higher priority than the current priority at the end of the rule firing. We use the program below with, for given n, the initial query `?- a(`n`).`

```
a(X) <=> X > 0 | a(X-1) pragma priority(1).
a(0) <=> true pragma priority(1).
```

Let P be the above program, P' the program without priority annotations, and $T(P)$ the result of applying our translation scheme of Section 5 to P.

Because of Theorem 3, every derivation of P' under the ω_r semantics corresponds to a derivation of P under the ω_p semantics. This allows evaluating the overhead introduced by our translation from P to $T(P)$ with respect to the execution of P' under the ω_r semantics. Indeed, the execution strategy (in terms of constraint activations and rule firings) is exactly the same in both P' and $T(P)$. Note that this program is a worst case scenario: the rules are single-headed, so no time is spent looking for partner constraints, and the only built-in constraint (`X > 0`) is cheap to check for implication. Moreover, the overhead generated by our source transformation is not compensated by any advantage that could otherwise follow from a different execution strategy.

By using its *late storage* analysis [18], the K.U.Leuven CHR compiler detects that in P', the `a`/1 constraints only need to be stored if none of the program rules apply. When applying this analysis on $T(P)$, it is confused by the explicit execution control mechanism and therefore it assumes that (the extended versions of) the `a`/1 constraints always need to stored as soon as they are asserted (which is in fact not the case). For a fair comparison, we have turned off the storage analysis so that constraint storage is equal in both P' and $T(P)$.

We have measured the runtime relative to the execution of P' under the refined operational semantics without the late storage analysis. For the ω_p semantics, execution of the program resulting from using $T(P)$[9] is about 14.3 times slower than execution of P' under the ω_r semantics. A hand optimized version in which we have used global variables instead of constraints to represent the current priority, and optimized the scheduling and reactivation mechanism, is still about 4.25 times slower.[10] The hand optimizations can easily be automated, though not at the level of the source-to-source transformation. They give a first indication of what is reachable in terms of efficiency when compiling CHR$^{\mathrm{rp}}$ programs directly into the host language.

8. Conclusions

We have extended the Constraint Handling Rules language with user-defined rule priorities. The extended language, called CHR$^{\mathrm{rp}}$, supports a high-level, flexible and declarative form of execution control. It allows the programmer to write programs that are more concise, but perhaps not confluent under the theoretical operational semantics ω_t of CHR, while still offering a high-level and declarative reading. The latter is in contrast with the low-level, procedural nature of execution under the refined operational semantics ω_r. In CHR$^{\mathrm{rp}}$, all execution control information is in the priority annotations, which causes a clear distinction between the logic and the control aspect of a CHR$^{\mathrm{rp}}$ program.

We have formalized the syntax and semantics of CHR$^{\mathrm{rp}}$ and investigated its theoretical properties. We then proposed a source-to-source transformation that allows CHR$^{\mathrm{rp}}$ programs with only *static* rule priorities to be executed under the ω_r semantics. It is shown that this transformation is correct and the runtime and space overhead it creates are examined. Experimental evaluation has demonstrated the advantage of static priorities over rule order under the ω_r semantics and determined the magnitude of the runtime overhead generated by the transformation.

[9] Although all rules fire at the highest priority, we have included the calls to `check_activation`/0 in the translation.

[10] We do still check for higher priority scheduled constraints at the end of each rule body though.

8.1 Related Work

8.1.1 Rule Priorities

Rule priorities are found in many rule based languages. Production rule systems like CLIPS [17], Jess [12] or JBoss Rules [23] use rule priorities (*salience*) as part of conflict resolution. These priorities are either integers, or a partial order between rules as in the *active database system* Starburst [34]. Most production rule systems use the RETE matching algorithm [11], which is an eager matching algorithm that exhibits high memory requirements, but allows an easy implementation of priority schemes. A lazy matching algorithm called LEAPS [21] is used by a few production rule systems, such as Venus [5] and JBoss Rules (in an experimental stadium). This algorithm is similar to what is used by CHR implementations based on the refined operational semantics. It seems that in these systems, priorities are only used relative to the active constraint (dominant object in LEAPS terminology), thus not allowing "global" priorities like the ones presented in this paper.

Priorities have also been introduced in term rewriting systems ("Priority Rewrite Systems" [2]). There, rule priorities are used to resolve conflicts that lead to non-confluent behavior. More recently, this idea has also been applied to term-graphs [6]. In [4] rule priorities are used to choose between alternative answer sets in answer set programming and in [16] priorities are used for a similar purpose in the context of defeasible logic programming.

A bottom-up logic programming language with prioritized rules is presented in [15]. The language supports dynamic priorities that depend on the first head in the rule. It only computes those (partial) matchings that have the current highest priority, but stores them in a RETE-like fashion.

8.1.2 Priorities in C(L)P Systems

Many Constraint (Logic) Programming systems offer some form of priorities (see [30]). The SICStus finite domain solver CLP(FD) [7] uses two priority levels: the highest priority is reserved for constraint propagators implemented by *indexicals*. Specialized algorithms implementing global constraints are scheduled at the lowest priority.

ILOG CPLEX [19] uses priorities to select variables for branching during branch and bound optimization. ILOG Solver [20] appears to be using only one *propagation queue* although its manual suggests that constraints can be pushed onto a *constraint priority queue* at a given priority.

The ECL^{i}PS^e Constraint Logic Programming system [33] supports execution at 12 different priorities. A goal G is executed at a given priority P by using `call_priority(`G`,`P`)`. The priority system is shared by all constraint libraries, which allows a form of global control over cooperating constraint solvers. The Extended Constraint Handling Rules library (`ech`) of ECL^{i}PS^e uses the priority system to support a form of static constraint priorities.

The execution strategy followed by `ech` is based on the concept of an active constraint, as is the case under the ω_r semantics. If a new CHR constraint is asserted in the body of a rule, it becomes active if it has a higher priority than the current active constraint, and otherwise it is scheduled for activation at its own (lower) priority. If a built-in constraint wakes up a set of CHR constraints, these are activated in priority order or scheduled for activation if their priority is lower than that of the current active constraint. See [8] for more details.

8.2 Future Work

A source-to-source transformation has the advantage that all analyses and optimizations of the current CHR compilers can be applied to the transformed program. However, often the analyses are too general to be able to detect opportunities for optimization of

programs resulting from the transformation. For example, the observation analysis of [29] which is used to detect whether or not a CHR constraint should be stored, cannot deal with the complex reactivation policy implemented by the source transformation, and therefore assumes the worst (i.e., all constraints are always observable). A direct compilation of CHR^{rp} programs clearly supports more optimizations at the level of priorities.

We plan to investigate how our implementation based on lazy matching can be extended towards dynamic rule priorities. So far, we have scheduled each new or reactivated constraint at each priority at which it has an occurrence. Clearly, in the case of dynamic rule priorities this is not always feasible. A solution to this problem could be to store those partial matches that determine the rule priority (in general there may be multiple subsets of the heads that fully determine the priority). Whenever a constraint whose arguments do not appear in a dynamic priority formula is (re-)activated, it is scheduled for each priority-determining partial match.

The implementation based on a scheduling system (in this case a priority queue) for activating constraints and batch mode processing of rule bodies, can be the basis for other forms of execution control like execution in phases or probabilistic CHR [14].

References

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *3rd Intl. Conf. on Principles and Practice of Constraint Programming*, volume 1330 of *LNCS*, pages 252–266, 1997.

[2] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. Term rewriting systems with priorities. In *2nd Intl. Conf. on Rewriting Techniques and Applications*, volume 256 of *LNCS*, pages 83–94, 1987.

[3] H. Betz and T. W. Frühwirth. A linear-logic semantics for constraint handling rules. In *11th Intl. Conf. on Principles and Practice of Constraint Programming*, volume 3709 of *LNCS*, pages 137–151, 2005.

[4] G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. *Artif. Intell.*, 109(1-2):297–356, 1999.

[5] J. C. Browne, E. A. Emerson, M. G. Gouda, D. P. Miranker, et al. A new approach to modularity in rule-based programming. In *6th Intl. Conf. on Tools with Artificial Intelligence*, pages 18–25. IEEE Computer Society, 1994.

[6] R. Caferra, R. Echahed, and N. Peltier. Rewriting term-graphs with priority. In *8th ACM SIGPLAN Symp. on Principles and Practice of Declarative Programming*, pages 109–120, 2006.

[7] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *9th Intl. Symp. on Programming Languages: Implementations, Logics, and Programs*, volume 1292 of *LNCS*, pages 191–206, 1997.

[8] L. De Koninck, T. Schrijvers, and B. Demoen. CHR^{rp}: Constraint Handling Rules with rule priorities. Technical Report 479, K.U.Leuven, Belgium, March 2007.

[9] G. J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, University of Melbourne, Victoria, Australia, Dec 2005.

[10] G. J. Duck, P. J. Stuckey, M. García de la Banda, and C. Holzbaur. The refined operational semantics of Constraint Handling Rules. In *20th Intl. Conf. on Logic Programming*, volume 3132 of *LNCS*, pages 90–104, 2004.

[11] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artif. Intell.*, 19(1):17–37, 1982.

[12] E. Friedman-Hill. *JESS 7.0p1: The rule engine for the Java platform*. http://herzberg.ca.sandia.gov/jess.

[13] T. W. Frühwirth. Theory and practice of Constraint Handling Rules. *J. Log. Program.*, 37(1-3):95–138, 1998.

[14] T. W. Frühwirth, A. Di Pierro, and H. Wiklicky. Probabilistic constraint handling rules. *Electr. Notes Theor. Comput. Sci.*, 76, 2002.

[15] H. Ganzinger and D. A. McAllester. Logical algorithms. In *18th Intl. Conf. on Logic Programming*, volume 2401 of *LNCS*, pages 209–223, 2002.

[16] A. J. García and G. R. Simari. Defeasible logic programming: An argumentative approach. *Theory Pract. Log. Program.*, 4(1-2):95–138, 2004.

[17] J. C. Giarratano. *CLIPS User's Guide, Version 6.20*, 2002. http://www.ghg.net/clips/CLIPS.html.

[18] C. Holzbaur, M. J. García de la Banda, P. J. Stuckey, and G. J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. *CoRR*, cs.PL/0408025, 2004.

[19] ILOG. *ILOG CPLEX 7.5: Reference Manual*, 2001.

[20] ILOG. *ILOG Solver 5.1: Reference Manual*, 2001.

[21] D. P. Miranker, D. A. Brant, B. Lofaso, and D. Gadbois. On the performance of lazy matching in production systems. In *8th National Conf. on Artificial Intelligence*, pages 685–692. AAAI Press / The MIT Press, 1990.

[22] F. Morawietz. Chart parsing and constraint programming. In *18th Intl. Conf. on Computational Linguistics*, pages 551–557. Morgan Kaufmann, 2000.

[23] M. Proctor, M. Neale, P. Lin, and M. Frandsen. *Drools Documentation, Version 3.0.5*, 2006. http://www.jboss.com/products/rules.

[24] G. Ringwelski and M. Hoche. Impact- and cost-oriented propagator scheduling for faster constraint propagation. In *19th Workshop on (Constraint) Logic Programming*, volume 2005-01 of *Ulmer Informatik-Berichte*, pages 88–98. Universität Ulm, Germany, 2005.

[25] T. Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, Jun 2005.

[26] T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: Implementation and application. In *First Workshop on Constraint Handling Rules: Selected Contributions*, volume 2004-01 of *Ulmer Informatik-Berichte*, pages 1–5. Universität Ulm, 2004.

[27] T. Schrijvers et al. The Constraint Handling Rules home page, 2007. http://www.cs.kuleuven.be/~dtai/projects/CHR/.

[28] T. Schrijvers and T. W. Frühwirth. Optimal union-find in Constraint Handling Rules. *Theory Pract. Log. Program.*, 6(1-2):213–224, 2006.

[29] T. Schrijvers, P. J. Stuckey, and G. J. Duck. Abstract interpretation for Constraint Handling Rules. In *7th ACM SIGPLAN Symp. on Principles and Practice of Declarative Programming*, 2005.

[30] C. Schulte and P. J. Stuckey. Speeding up constraint propagation. In *10th Intl. Conf. on Principles and Practice of Constraint Programming*, volume 3258 of *LNCS*, pages 619–633, 2004.

[31] J. Sneyers, T. Schrijvers, and B. Demoen. The computational power and complexity of Constraint Handling Rules. In *2nd Workshop on Constraint Handling Rules*, volume 421 of *Reports CW*, pages 3–17. Dept. of Computer Science, K.U.Leuven, Belgium, 2005.

[32] 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 Constraint Handling Rules*, volume 421 of *Reports CW*, pages 47–62. Dept. of Computer Science, K.U.Leuven, Belgium, 2005.

[33] M. Wallace, S. Novello, and J. Schimpf. ECL^{i}PS^e: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, 1997.

[34] J. Widom. The Starburst rule system. In *Active Database Systems: Triggers and Rules For Advanced Database Processing*, pages 87–109. Morgan Kaufmann, 1996.