Antimonotony-based Delay Avoidance for CHR

Tom Schrijvers, Bart Demoen

Report CW 385, July 2004



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

Antimonotony-based Delay Avoidance for CHR

Tom Schrijvers, Bart Demoen

Report CW 385, July 2004

Department of Computer Science, K.U.Leuven

Abstract

We present an optimisation for Constraint Handling Rules (CHR) that reduces the amount of variables a constraint delays on. This optimisation reduces the overhead of delaying on variables as well as the needless reawakening of delayed constraint.

A correctness proof of the optimisation based on the refined operational semantics establishes the validity of the optimisation and sheds light on possible extensions as well as related optimisations.

The delay avoidance optimisation shows good speedups for some programs and no noticable slowdown for others.

Antimonotony-based Delay Avoidance for CHR

Tom Schrijvers*
Bart Demoen
{toms,bmd}@cs.kuleuven.ac.be

Abstract

We present an optimisation for Constraint Handling Rules (CHR) that reduces the amount of variables a constraint delays on. This optimisation reduces the overhead of delaying on variables as well as the needless reawakening of delayed constraint.

A correctness proof of the optimisation based on the refined operational semantics establishes the validity of the optimisation and sheds light on possible extensions as well as related optimisations.

The delay avoidance optimisation shows good speedups for some programs and no noticable slowdown for others.

1 Introduction

CHR is a simple rule-based forward-chaining language originally intended for writing constraint solvers. The initial operational semantics of the langauge, given in [3], contained a lot of indeterminism. However, the actual operational semantics shared by CHR implementations are much less indeterministic and many general purpose applications have been written for them.

We refer the reader to [4] for an overview of CHR and related concepts. Several implementations of CHR exist, embedded in different host languages. Prolog is the main host language, with three main implementations we are aware of, but systems also exist for Java and Haskell.

Recently discussion has started on the adoption of a common CHR compiler to be shared among different systems. In this way, all system based on this common compiler will benefit from improvements to it.

In this paper we present a particular optimisation technique for CHR programs which we call antimonotony-based delay avoidance. It is based on static analysis and aims at avoiding the rechecking of the program rules for constraints when it is unnecessary. We will show that this technique gives good speedups in particular cases.

The outline of the rest of the paper is as follows. First, in Section 2, we present some properties and initial assumptions related to the refined operational semantics and programs under consideration. Next, in Section 3, we present the actual optimisation proposed in this paper. A correctness proof of this optimisation is given in Section 4 and the actual implementation of this optimisation in the K.U.Leuven CHR system [5] is discussed in Section 5. The implementation is evaluated in Section 6. Finally, we mention some related and future work in Section 7 and conclude in Section 8.

2 The Refined Operational Semantics of Constraint Handling Rules

Before we present the antimonotony-based delay avoidance optimisation and its correctness proof, we first establish the operational semantics that are considered in this paper.

We refer the reader to [1] for the original version of the refined operational semantics for CHR. We will deviate from that version in two ways. Firstly, we only consider CHR programs in Head Normal Form and secondly, we infer certain properties of constraint projection and fixed variables that will be of use. These two issues are discussed in Section 2.1 and Section 2.2 respectively. In Section 2.3 we give a definition of the antimonotony property that will be essential to formulate the delay avoidance optimisation.

*Research Assistant of the fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen)

2.1 Head Normal Form

For reasons of simplicity, we will only consider CHR programs in Head Normal Form, a form that makes the guards explicit. A CHR program P is in Head Normal Form if every rule r in P is in Head Normal Form. We say that a CHR rule r is in Head Normal Form if all arguments of constraints in the head are variables and different from other arguments.

It is possible to transform every CHR program into an equivalent program in Head Normal Form. A procedure for this is given in [2].

Example 2.1 For example, the following program is not in Head Normal Form:

```
r1 @ fibonacci(N,M1) \ fibonacci(N,M2) <=> M1 = M2.
r2 @ fibonacci(0,M) ==> M = 1.
r3 @ fibonacci(1,M) ==> M = 1.
r4 @ fibonacci(N,M) ==> N > 1 |
                N1 is N-1, fibonacci(N1,M1),
                N2 is N-2, fibonacci(N2,M2),
                M is M1 + M2.
```

because rules $r1$, $r2$ and $r3$ are not in Head Normal Form. In rule $r1$ the same variable, N, appears twice in the head and in rules $r2$ and $r3$ two arguments are not variables, but numbers.

Example 2.2 However, the following program is in Head Normal Form and equivalent to the first:

```
r1 @ fibonacci(N1,M1) \ fibonacci(N2,M2) <=> N1 == N2 | M1 = M2.
r2 @ fibonacci(N,M) ==> N == 0 | M = 1.
r3 @ fibonacci(N,M) ==> N == 1 | M = 1.
r4 @ fibonacci(N,M) ==> N > 1 |
                N1 is N-1, fibonacci(N1,M1),
                N2 is N-2, fibonacci(N2,M2),
                M is M1 + M2.
```

2.2 Constraint Projection and Fixed variables

We will derive here a number of properties of fixed variables as defined in the refined operational semantics [1]. These properties will facilitate the correctness proof.

In the operational semantics, only the **Solve** rule refers to fixed variables. When a builtin constraint c_b is added to the builtin constraint store, all constraints have to be put on the execution stack, execpt those constraints whose variables are all fixed.

Intuitively, fixed variables are variables that cannot be further constrained. Hence the applicability of a rule cannot change for a constraint with all fixed arguments by a builtin constraint. E.g. in the Herbrand constraint domain, fixed variables are ground variables.

We will formulate the defintion of fixed variables in terms of constraint projection. Constraint projection is denoted as $\bar{\exists}_A F$ in [1]. Here we will use a slightly different notation: $\pi_{vars(A)}(F)$[1]. It stands for $\pi_V(F) = \exists X_1, ..., \exists X_n : F$ with $vars(F) \backslash V = \{X_1, ..., X_n\}$.

Based on this definition of projection, fixed variables are defined as follows in [1]:

$$v \in fixed(B) \Leftrightarrow \mathcal{D} \models \pi_v(B) \wedge \forall \rho : \pi_{\rho(v)}(\rho(B)) \rightarrow v = \rho(v) \tag{1}$$

with ρ any variable substitution. The following property holds for fixed variables:

$$V \subseteq fixed(B) \implies fixed(B) \subseteq fixed(B \wedge B') \tag{2}$$

$$V \subseteq fixed(B) \implies (B \wedge \pi_{vars(B \wedge B') \backslash V'}(B \wedge B') \Leftrightarrow B \wedge B') \tag{3}$$

[1]The function $var(A)$ returns the unquantified variables appearing in A.

with B' any conjunction of builtin constraints such that $\mathcal{D} \models B \wedge B'$ holds.

The former, property (2), holds since for any $v \in fixed(B)$:

$$\begin{array}{rcl} \mathcal{D} \models B \wedge B' & \Longrightarrow & \pi_v(B \wedge B') \\ \forall \rho : \pi_{\rho(v)}(\rho(B)) \rightarrow v = \rho(v) & \Longrightarrow & \forall \rho : \pi_{\rho(v)}(\rho(B \wedge B')) \rightarrow v = \rho(v) \end{array}$$

The latter, property (3), holds because for any $v \in fixed(B)$ it holds that $v \in fixed(B\ wedge B')$. Now, in $\pi_{vars(B \wedge B') \setminus V'}(B \wedge B')$ such a v is existentially quantified. This existential quantification corresponds to a variable substitution ρ. Since v is fixed however, it holds that $\rho(v) = v$. Hence it is allowed to drop the existential quantification.

2.3 Antimonotony

Antimonotony is an interesting property, somewhat similar to fixed variables, regarding the entailment of a conjunction of constraints. This concept will be essential for the formulation of the delay optimisation.

Definition 2.3 We say that a conjunction of builtin constraints B is antimonotone in a set of variables V if and only if:

$$\forall B_1, B_2 : (\pi_{vars(B) \setminus V}(B_1 \wedge B_2) \Leftrightarrow \pi_{vars(B) \setminus V}(B_1)) \Rightarrow ((\mathcal{D} \not\models B_1 \rightarrow B) \Rightarrow (\mathcal{D} \not\models B_1 \wedge B_2 \rightarrow B))$$

where B_1 and B_2 are any conjunction of builtin constraints.

It is clear from the definition that if B is antimonotone in a set V, then B is also antimonotone in any subset of B.

Example 2.4 For example, for the Herbrand constraint domain, the constraint $var(X)$ that constrains a variable X to be not bound to a term is antimonotone in $\{X\}$. Since any conjunction of constraints that does not entail $var(X)$ must bind X to some term, it is clear that no additional constraint can undo the binding.

Based on the above defintion for antimonotony, we can define antimonotony of a CHR program P in $c[i]$, the ith argument of a constraint c.

Definition 2.5 We say that a CHR program P is antimonotonous in $c[i]$, the ith argument of a constraint c, if and only if, for every occurence j of the constraint c in program P the guard of the corresponding rule is antimonotone in $\{c[i]\#j\}$.

Example 2.6 The CHR program in Example 2.2 is not antimonotone in $fibonacci[1]$ since all the guards in the program are not antimonotone in it. However, the program is antimonotone in $fibonacci[2]$ for the Herbrand constraint domain as it is defined in Prolog, since none of the guards mentions it. Hence any constraint, i.e. unification that affects $fibonacci[2]$ does not affect the applicability of the guard.

3 Delay Avoidance

In this Section, we discuss an optimisation that attempts to reduced the number of constraints that need to be triggered in the **Solve** transition rule. The optimisation applies to all constraint arguments in which the applicability of the rules in program P is antimonotonic.

The optimisation consists of replacing the **Solve** rule with the following variant:

1. **Solve'**. $\langle [c_b|A], S_1 \uplus S_0, B, T \rangle_n \rightarrowtail \langle S_1 +\!\!+ A, S_1 \uplus S_0, B \wedge c_b, T \rangle_n$ where c_b is a builtin constraint, and for every constraint $c \in S_0, \exists V_1, V_2 : vars(c) = V_1 \cup V_2 \wedge V1 \subseteq fixed(B) \wedge$ all variables in V_2 only appear in arguments of c that are antimonotonic in P.

4 Correctness Proof

Theorem 4.1 *For any state σ:*

$$\sigma = \langle [c_b|A], S, B, T\rangle_n$$

*where c_b is a builtin constraint, applying the **Solve** rule or applying the mofied rule **Solve'** can lead to the same final state given that transition was started from an initial state.*

Proof Consider such a state $\sigma = \langle [c_b|A], S_0 \uplus S_1 \uplus S_2, B, T\rangle_n$ where:

- for every constraint $c \in S_0 : vars(c) \subseteq fixed(B)$.
- for every constraint $c \in S_2, \exists V_1, V_2 : vars(c) = V_1 \cup V_2 \wedge V_1 \subseteq fixed(B) \wedge$ all variables in V_2 only appear in arguments of c that are antimonotonic in P.

By applying the **Solve** rule, we obtain a state σ_1:

$$\sigma_1 = \langle (S_1 \uplus S_2) +\!\!+ A, S_0 \uplus S_1 \uplus S_2, B \wedge c_b, T\rangle_n$$

Alternatively, by applying the **Solve'** rule we obtain:

$$\sigma_1' = \langle S_1 +\!\!+ A, S_0 \uplus S_1 \uplus S_2, B \wedge c_b, T\rangle_n$$

Both rules leave unspecified in what order constraints should be added to the constraint store. Here we chose a particular ordering and chose that for this ordering both states σ_1 and σ_1' will lead to the same final state, if for all inderministic choice the same choice is made in both transition sequences. The particular ordering we will consider is the following: all elements of S_1 should be put on the execution stack before those of S_2. This means:

$$\sigma_1 = \langle S_1 +\!\!+ S_2 +\!\!+ A, S_0 \uplus S_1 \uplus S_2, B \wedge c_b, T\rangle_n$$

and additionally we chose the same ordering of S_1 in σ_1 and σ_1'.
Now, if σ_1 reaches a final state, then there must be a finite transition sequence $\sigma_1 \rightarrowtail^* \sigma_l$ where

$$\sigma_l = \langle S_2 +\!\!+ A, S_l, B_l, T_l\rangle_{n_l}$$

and it is the first such state in the deriviation. Since all transition rules only consider the topmost element of the execution stack, it is possible to apply the same transition steps to σ_1' to reach a state σ_l':

$$\sigma_l' = \langle A, S_l, B_l, T_l\rangle_{n_l}$$

We will now show that σ_l' is the only state reachable from σ_l and that it is effectively reached. Name the constraints in S_2 according to their order on the execution stack respectively $c_1, c_2, ..., c_k$. Hence, the execution stack of $\sigma_l = [c_1, c_2, ..., c_k|A]$. We call $A_i = [c_i, ..., c_k|A]$ for $i = 1..k$ and $A_{k+1} = A$. We will show that $s_l = s_{l,1} \rightarrowtail^* s_{l,2} \rightarrowtail^* .. \rightarrowtail^* s_{l,k} \rightarrowtail^* s_{l,k+1} = s_l'$where $s_{l,i} = \langle A_i, S_l, B_l, T_l\rangle_{n_l}$ is the only possible transition sequence.

Now consider every transition sequence $\sigma_{l,i} \rightarrowtail^* \sigma_{l,i+1}$ for $i = 1..k$ separately. We will show that this transition sequence is the only possible one and that it looks like:

$$\sigma_{l,i} = \sigma_{l,i,0} \rightarrowtail \sigma_{l,i,1} \rightarrowtail ... \rightarrowtail \sigma_{l,i,o+1} \rightarrowtail \sigma_{l,i,o+2} = \sigma_{l,i+1}$$

with $\sigma_{l,i,j} = \langle [c_i : j|A_i + 1], S_l, B_l, T_l\rangle_{n_l}$ for $j = 1..o + 1$ and o the maximal occurrence number of constraint c_i in Program P.

- Consider $j = 0$. It is clear that in this state, only the **Reactivate** rule applies. It brings state $\sigma_{l,i,0}$ into state $\sigma_{l,i,1}$.

- Consider $j = o + 1$. It is clear that in this state, only the **Drop** rule applies. It brings state $\sigma_{l,i,o+1}$ into state $\sigma_{l,i,o+2}$.
- Consider $0 < j < o + 1$. We will show that only the **Default** rule applies in these states. The **Default** rule brings state $\sigma_{l,i,j}$ into state $\sigma_{l,i,j+1}$.

  Clearly, only one of the **Simplify**, **Propagate** or **Default** rules applies in state $\sigma_{l,i,j}$. If neither of the first two applies, then clearly the latter does.

  - Assume j corresponds to an occurrence of c_i after the backslash in a rule r. Then only the **Simplify** rule would apply.
  - Assume j corresponds to an occurrence of c_i before the backslash in a rule r. Then only the **Propagate** rule would apply.
  - Assume $c_i \in S_l$. This is necessary for either rule to apply.
  - Assume the necessary partner constraints are present in S_l. This is necessary for either rule to apply.
  - Assume the corresponding tuple is not present in the tuple history T_l. This is necessary for either rule to apply.

  Consider the earliest state σ_e where all partner constraints and c_i are present in the constraint store. A derivation must have started from this state where none of the constraints was removed, since they still are in the store. Hence the corresponding occurrence of the last constraint in rule r must have been at the top of the execution stack. The rule cannot have applied at that time, since the tuple does not appear in T_l. Hence, either the matching substitution did not exist or the guard was not entailed.

  Since the matching substitution exists now and the guard is entailed now, additional builtin constraints must have been added to the builtin constraint store after that point. Consider the state where the last such builtin constraint was added and both matching substitution started to exist and the guard was entailed. Either this was before σ or after σ_1, but then the **Solve** or **Solve'** rule would have been applied, one of the partner constraints would have been added, the rule r would have applied and the corresponding tuple would be present in T_l. Hence, the last builtin constraint must have been c_b, added in σ_1. Now, none of the partner constraints could have been added to the execution stack in S_1 because of c_b, since then the rule would have applied through that constraint and the tuple would have been present in T_l.

  Hence, for all partner constraints, either they are in S_2 or S_0. We know a matching substitution exists and the guard is entailed in $\sigma_{l,i,j}$. Hence, $\mathcal{D} \models B_{l,r} \wedge c_b \rightarrow (g \wedge \theta)$ and $\mathcal{D} \not\models B_{l,r} \rightarrow (g \wedge \theta)$ where $B_l = B_{l,r} \wedge c_b$.

  Consider all variables in the constraints. They are either fixed or appear in antimonotonic arguments. Call the sets of these variables respectively V_0 and V_2.

  Now, $\pi_{V_2}(B_{l,r} \wedge c_b)$ only depends on variables in V_2. Applying the defintion of antimonotony and based on what was established before, it holds that:

  $$D \not\models B_{l,r} \wedge \pi_{V_2}(B_{l,r} \wedge c_b) \rightarrow (g \wedge \theta)$$

  Since the variables in V_0 are fixed, applying property (3), this is equivalent with:

  $$D \not\models B_{l,r} \wedge c_b \rightarrow (g \wedge \theta)$$

  However, this conflicts with the prior assumption. Hence, the prior assumption was wrong and the **Simplification** or **Propagation** rule cannot apply.

Notice that the above proof also establishes the correctness or the **Solve** rule with respect to the following, more general **Solve"** rule:

1. **Solve"**. $\langle [c_b|A], S, B, T\rangle_n \rightarrowtail \langle S{+\!\!+}A, SS, B \wedge c_b, T\rangle_n$ where c_b is a builtin constraint.

5 Implementation

We have implemented the delay avoindance optimisation in the hProlog branch of the K.U.Leuven CHR system [5]. The constraint domain used in this system is the standard Prolog Herbrand constraint domain with term unification as the main constraint.

Instead of limiting the optimisaton for programs in Head Normal Form or transforming the program to this form, the implementation deals with matchings in the head directly. Any non-variable argument corresponds with a unification in the guard, which is not antimonotone. The same holds for a variable that appears more than once in the head.

In addition, the implementation has limited knowledge of antimonotony for builtin constraints:

- If a variable does not occur in a goal, then the goal is antimonotone in that variable. This covers `true/0`.
- `var/1` is antimonotone in its argument.
- A conjunction (disjunction) is antimonotone in a variable if its conjuncts (disjuncts) are antimonotone in the variable.

The optimisation is handled conservatively for other guards, i.e. if it cannot prove that a guard is antimonotone in a particular constraint argument, it assumes that it is not anitmonotone. In particular any user defined predicate that appears in the guard is assumed to not be antimonotone in any of its arguments.

In the standard implementation, when a constraint delays, *all* variables in the constrain are retrieved. The constraint is then stored in an attribute[2] for all variables. When any of the variables is unified, the constraint is retrieved and triggered.

Based on the analysis of antimonotone arguments, the variable retrieval operation is replaced by a specialised operation for each argument. That specialised operation only considers arguments that are not antimonotone. This avoids the triggering of the constraint when any of variables in antimonotone arguments is unified.

Example 5.1 Consider again example 2.2. The standard procedure to retrieve all the delay variables would be:

```
delay_variables(Constraint,Vars) :-
        term_variables(Constraint,Vars).
```

However, by inferring that the `fibonacci/2` constrait is antimonotone in its second argument, the following specialized code can be generated:

```
delay_variables(fibonacci(N,_),Vars) :-
        term_variables(N,Vars).
```

Since in practice the first argument to `fibonacci/1` is always ground, the above code will alway unify `Vars` with `[]` and effectively the constraint is not delayed on any variable.

6 Experimental Evaluation

To experimentally evaluate the antimonotony-based optimisation described in the previous section, we consider the effect on the runtime of our standard set of CHR benchmarks [**?**] together with two variants of the `fibonacci` benchmark which differ in their furst rule.

The `fibonacci` program in the standard benchmark is the most optimized one. Its first rule is:

```
r1 @ fibonacci(N,M1) # ID \ fibonacci(N,M2) <=> var(M2) | M1 = M2 pragma passive(ID).
```

[2] Attributed variables are used in the Prolog implementation to store information on variables and to provide a unification handler

Benchmark	Unoptimized		Optimized	
bool	850	100.0%	880	103.5%
leq	1,210	100.0%	1,210	100.0%
zebra	1,690	100.0%	1,680	99.4%
primes	1,240	100.0%	1,240	100.0%
ta	1,340	100.0%	1,340	100.0%
wfs	940	100.0%	930	98.9%
fib	1,250	100.0%	1,230	98.4%
fibonacci	1,860	100.0%	1,000	53.8%
fibonacci1	2,910	100.0%	1,360	46.7%
fibonacci2	2,900	100.0%	1,360	46.9%
lookup	1,000	100.0%	780	78.0%

Table 1: Absolute and relative runtime of unoptimized vs. optimized benchmarks

The first rule of `fibonacci1` is, similar to example 2.2, but denormalized:

```
r1 @ fibonacci(N,M1) \ fibonacci(N,M2) <=> M1 = M2.
```

Finally, `fibonacci2` has the following first rule:

```
r1 @ fibonacci(N,M1) \ fibonacci(N,M2) <=> var(M2) | M1 = M2.
```

The standard `fib` benchmark differs from `fibonacci1` in that it uses a simplification rule. Because this is much more inefficient, this benchmark computes a smaller Fibonacci number.

In addition to the above standard benchmarks, we have also looked at the effect on the following CHR idiom:

```
entry(Key,Value) \ lookup(Key,Query) <=> Query = Value.
```

In the above rule, the both the `entry/2` and `lookup/2` constraints are antimonotonic in their second argument. The benchmark based on this idiom is called `lookup`: it consists of asserting and entry, immediately followed by a lookup.

Table 1 lists the runtime results in milliseconds of running the benchmarks in hProlog. The results clearly indicate that there is hardly any effect on the benchmarks where no static optimization is possible (`bool`, `leq` and `zebra`) or where dynamically no variables occur (`primes`, `ta` and `wfs`).

In the `fib` benchmark the optimization does have some effect, but it is not manifest. The reason is that the inherent inefficiency of the simplification rule is predominant. However, in all three variants of the `fibonacci` benchmark, the runtime is about halved by the delay-avoidance optimization. Similarly, for the `lookup` benchmark there is a noticable speedup, about 20%.

7 Future and Related Work

Several variants or extensions to the proposed optimisation are possible. We list several here:

7.1 Monotony-based Delay Avoidance

In the above we have inferred the antimonotony property based on the statically for the variables in the Head Normal Form of the program. We could also have look at the monotony property, i.e.

Definition 7.1 We say that a conjunction of builtin constraints B is monotone in a set of variables V if and only if:

$$\forall B_1, B_2 : (\pi_{vars(B)\setminus V}(B_1 \wedge B_2) \Leftrightarrow \pi_{vars(B)\setminus V}(B_1)) \Rightarrow ((\mathcal{D} \models B_1 \rightarrow \pi_V(B)) \Rightarrow (\mathcal{D} \models B_1 \wedge B_2 \rightarrow \pi_V(B))$$

where B_1 and B_2 are any conjunction of builtin constraints.

This property captures those variables that do not further affect the applicability of the guard once they are sufficiently constrained.

Example 7.2 A trivial example of this is `true`, which is monotone in any variable and entailed by any builtin constraint store.

Example 7.3 Term equality, which for example occurs in the matching in the head of a rule, is monotone in the involved variables. Once two variables are equal or a variable is equal to a term with a particular functor, this equality cannot be undone anymore by additional builtin constraints.

There is an overlap between the antimonotone and the montone delay avoidance: those variables which do not influence the entailment of the guard are covered by both approaches.

The usefulness of the monotony property for optimisation is the following: consider that the guard of a rule is not entailed, but that the projection on the variables it is monotone in is entailed. Then any further constraining of those variables will not alter the applicability of the rule.

It may seem hard to verify the entailment of the projection of the guard in practice for an arbitrary constraint domain. However, for the Herbrand constraint domain, i.e. for term unification, it is fairly easy to check for the proper instantiation. In addition, a practical implementation may only check for this property the first time it delays the constraint instead of each time.

7.2 Wake conditions

The work in this paper is strongly related to the work by Duck et al. presented in [1]. They infer wake conditions from the guards which are supported by builtin constraint solvers. If a particular wake condition of a constraint is met by a builtin constraint, the constraint is woken. If none of the conditions is satisfied by a builtin constraint, the constraint need not be woken.

However, no formal relationship with the refined operational semantics or a correctness proof of the wake conditions approach has been given.

The approach in [1] also specialises waking of constraints to only recheck constraint occurrences which may have been affected by the builtin constraint. The same would be possible for the delay avoidance, i.e. for a builtin constaint only recheck the occurrences of constraints of which the affected variables are not antimonotone in the occurrence. However, to still abide the refined operational semantics, such a specialisation should be more conservative then in [1]. Indeed, the specialisation in [1] may recheck rules out of order, which is not allowed by the refined operational semantics.

Future work will investigate what the repercussion of this per-occurrence approach are on the standard Prolog compilation scheme of CHR and how it woud still be possible to satisfy the refined operational semantics.

8 Conclusion

We have presented in this paper a new optimisation for CHR based on the antimonotony property. The correctness of this optimisation has been proven based on the refined operational semantics of CHR. It was shown that the optimisation allows for good speedups in particular cases and the relationship with montone delay avoidance and wake conditions has been established.

References

[1] G. J. Duck, P. J. Stuckey, M. G. de la Banda, and C. Holzbaur. The real operational semantics of constraint handling rules. In *Proceedings of the 20th International Conference on Logic Programming*, 2004.

[2] G. J. Duck, P. J. Stuckey, M. Garcia de la Banda, and C. Holzbaur. Extending arbitrary solvers with constraint handling rules. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 79–90. ACM Press, 2003.

[3] T. Frühwirth. Constraint Handling Rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, number 910 in LNCS, pages 90–107. Springer Verlag, March 1995.

[4] T. Frühwirth. Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot, editors, *Special Issue on Constraint Logic Programming*, volume 37, October 1998.

[5] T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: implementation and application. In T. Frühwirth and M. Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, number 2004-01, 2004. ISSN 0939-5091.