Automatic Implication Checking for CHR Constraints

Tom Schrijvers [a,⋆] Bart Demoen [a] Gregory Duck [b]
Peter Stuckey [b,c] Thom Frühwirth [d]

[a] *Department of Computer Science, K.U.Leuven, Belgium*

[b] *Department of Computer Science, University of Melbourne, Australia*

[c] *NICTA Victoria Laboratory, Australia*

[d] *Faculty of Computer Science, University of Ulm, Germany*

Abstract

Constraint Handling Rules (CHRs) are a high-level rule-based programming language commonly used to define constraint solvers. We present a method for automatic implication checking between constraints of CHR solvers. Supporting implication is important for implementing extensible solvers and reification, and for building hierarchical CHR constraint solvers. Our method does not copy the entire constraint store, but performs the check in place using a trailing mechanism. The necessary code enhancements can be done by automatic program transformation based on the rules of the solver. We extend our method to work for hierarchically organized modular CHR solvers. We show the soundness of our method and its completeness for a restricted class of canonical solver as well as for specific existing non-canonical CHR solvers. We evaluate our trailing method experimentally by comparing with the copy approach: runtime is almost halved.

Key words: Constraint Handling Rules, implication checking, program transformation, constraint solver hierarchy

1 Introduction

Constraint handling rules [7] (CHRs) are a very flexible formalism for writing incremental constraint solvers and other reactive systems. In effect, the rules define transitions from one constraint set to an equivalent constraint set. Transitions serve to simplify constraints and detect satisfiability and unsatisfiability. CHRs have been used extensively (see e.g. [9]). Efficient implemen-

⋆ Research Assistant of the fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen) tations are already available for the languages SICStus Prolog and Eclipse Prolog, SWI-Prolog, HAL and Java. CHRs provide the best mechanism to date for creating executable user-defined constraint solvers.

In this paper we investigate how to automatically extend a CHR constraint solver to not only answer questions of satisfiability, but also to answer questions about *implication*. A constraint solver that supports implication can be used model more expressive constraints such as complex constraints formulas involving negation, conjunction and disjunction through reification. Many popular constraint solvers provide implication checking in one form or another, e.g. the *conditional* constraint combinator of Mozart [14] and the reified constraints of the `clp(FD)` library in SICStus [2]. Similarly implication allows the construction of hierarchical CHR solvers, where guards are defined by an underlying constraint solver implemented in CHRs, since an essential step in a CHR solver is to determine whether a guard in implied by the current store. Implication is also useful for building multiple cooperating solvers.

In this paper we extend a CHR solver to be able to answer *implication checks*, that is for a basic constraint c, is it implied by the current CHR constraint store. Implication checking for CHRs is an essential part of the Chameleon programming language [16], which uses CHRs for type class overloading (see [15]). We improve on the basic version of implication checking used by Chameleon.

Previously, it was already shown how to extend built-in solvers with implication checks to support CHR solvers in [4]. In this paper we have added a means to extend CHR solvers with other CHR solvers.

In [3] a general technique is presented for extending implication checks on basic constraints to implication checks of arbitrary logic formulas. The CHR implication checks presented in this paper can be extended to arbitrary formulas in that way.

The first related technique for CHR was already sketched in [6]. However, that technique does not represent an implication check, but rather a reified constraint: implication may not only succeed or fail but it may also delay until it can be decided. However, since higher level constraints are not removed, termination problems can be caused by the automatically generated rules. In that case the user has to modify the rules, which may cause more incompleteness.

2 CHR Solvers

2.1 CHR Syntax and Operational Semantics

In this subsection we briefly introduce the syntax and semantics of CHR programs or solvers. The other relevant aspects of CHR solvers are covered in the rest of this section. For a more extensive introduction and survey of CHR we refer the reader to [7].

Syntax: A CHR solver CS consists of a sequence of CHR rules. There are three kinds of CHR rules. We introduce them with an example.

Example 2.1 The following four CHR rules define an equality solver eq, with `eq/2` the equality constraint:

```
reflexive  @ eq(X,Y) <=> X == Y | true.
redundant  @ eq(X1,Y1) \ eq(X2,Y2) <=> X1 == X2, Y1 == Y2 | true.
symmetric  @ eq(X,Y) ==> eq(Y,X).
transitive @ eq(X1,Y1), eq(X2,Y2) ==> Y1 == X2 | eq(X1,Y2).
```

with the arguments of `eq/2` variable identifiers and `==/2` is a built in check for syntactic identity.

The `reflexive` rule is an example of a simplification rule. It states that we can replace the left hand side $eq(x, y)$ by the right hand side $true$ if the guard $x \equiv y$ holds, that is we can eliminate things of the form $eq(v, v)$. The `symmetric` rule is an example of a propagation rule. It states that when we have something matching the left hand side $eq(x, y)$ we should add the right hand side $eq(y, x)$. The `redundant` rule is an example of a simpagation (simplification and propagation) rule, it says if we have two constraints matching the left hand side $eq(x_1, y_1)$ and $eq(x_2, y_2)$ that satisfy the guard $x_1 \equiv x_2$ and $y_1 \equiv y_2$ we can eliminate the part matching after the \, that is, $eq(x_2, y_2)$. In other words we can eliminate duplicate eq constraints. The final rule is another propagation rule, which defines that equality is transitively closed.

The part before the `@` symbol is the name of the rule, it is optional. The part before the arrow is called the *head* of the rule. The part between the arrow and the pipe symbol (`|`) is called the *guard* of the rule (optional). The part after the pipe symbol is the *body* of the rule.

The head of a rule is a sequence of CHR constraints. The guard is a sequence of given built-in constraints. The body is a sequence of both CHR and built-in constraints.

A rule partially defines the constraints in the head. Built-in constraints are undefined, they are implemented in an already existing solver.

In general we can think of every CHR rule as a simpagation rule H_k \ H_r `<=>` G `|` B, where H_k and H_r are the sequences of head constraints, G is the guard and B is the body of the rule. If H_k is empty, the rule is a simplification rule and if H_r is empty, the rule is a propagation rule.

Operational Semantics: The operational semantics of CHR can be described as a state transition system [1]. Execution starts from an initial state σ_0 comprising the initial multi-set of constraints (also query or constraint store). A CHR rule application corresponds with a state transition $\sigma_i \rightarrowtail_{CS} \sigma_{i+1}$ and updates the multi-set of constraints. A rule is applicable if there are distinct constraints in the current state that match the head of the rule and if—under this matching—the guard is implied by the built-in constraints of the current state.

A terminating CHR derivation d is one that starts from an initial state σ_0 and reaches, after a finite number of transition steps, a final state σ. A final state σ is one in which no more transition step is possible. In general for the same initial state many different final states may be reached through many different derivations. We denote a particular derivation d from σ_0 to σ as $\sigma_0 \rightarrowtail_{CS}^{d} \sigma$. Also, we denote the final state obtained through a derivation d from an initial state σ_0 as $solve_d(\sigma_0)$. Note that CHR is a committed-choice language, i.e. rule applications cannot be undone. So a particular execution of a query will involve exactly one of the possible derivations.

In this paper we will use a particular instance of the semantics. The refined operational semantics [5] is the actual operational semantics implemented by most CHR systems, such as those in SICStus [8], HAL [10] and the K.U.Leuven CHR system [11]. These semantics describe a particular execution strategy that considerably reduces the number of different possible derivations for any initial state. CHR rules are applied in a top-to-bottom manner. The state is separated into two parts: a sequence of goals to be processed from left to right, and a multiset of constraints used for matching. Rules are applied using the notion of an *active* constraint, the last constraint added, which is exhaustively used in a matching before becoming inactive. For more details see [5].

Example 2.2 The following example informally describes a derivation under the refined operational semantics.

Consider the execution of the goal `eq(a,b), eq(b,c)` for the `eq/2` solver defined in Example 2.1.

Adding $eq(a,b)$ the `reflexive` rule is not applicable because a ==b does not hold, the `redundant` rule misses a second constraint, the `symmetric` rule adds $eq(b,a)$. This constraint adds $eq(a,b)$ using the `symmetric` rule, which is then deleted by the `redundant` rule. The `transitive` rule now can match $(eq(a,b), eq(b,a)$`==>`$b == b$ `|` $eq(a,a))$ so adds $eq(a,a)$ which is removed by the `reflexive` rule. Similarly the `transitive` rule adds $eq(b,b)$ which is deleted by the `reflexive` rule. The store is currently $\{eq(a,b), eq(b,a)\}$.

The addition of $eq(b,c)$ causes the addition of $eq(c,b)$ using the `symmetric` rule, and the `transitive` rule adds $eq(b,b)$ and $eq(c,c)$ which are immediately deleted by the `reflexive` rule, as well as $eq(c,a)$ $(eq(c,b), eq(b,a)$`==>`$b ==$ b `|` $eq(c,a))$ which adds $eq(a,c)$ using the `symmetric` rule. Later the `transitive` rule adds another copy of $eq(a,c)$ $(eq(a,b), eq(b,c)$`==>`$b == b$ `|` $eq(a,c))$ which is deleted using the `redundant` rule. The final store is $\{eq(a,b),\ eq(b,a),\ eq(b,c),\ eq(c,b),\ eq(a,c),\ eq(c,a)\}$

Part of the derivation is illustrated below, with the two parts of the state shown as $\langle Goal, Store\rangle$ and the active constraint in bold, and matching constraints underlined.

$$\begin{array}{lll}
 & & \langle [eq(a,b), eq(b,c)], \emptyset\rangle \\
 & \rightarrowtail_{CS} & \langle [eq(b,c)], \{\underline{eq(\mathbf{a},\mathbf{b})}\}\rangle \\
\texttt{symmetric} & \rightarrowtail_{CS} & \langle [eq(b,c)], \{eq(a,b), \underline{eq(\mathbf{b},\mathbf{a})}\}\rangle
\end{array}$$

$$\begin{array}{ll}
\texttt{symmetric} & \rightarrowtail_{CS} \langle [eq(b,c)], \{\underline{eq(a,b)}, eq(b,a), \underline{eq(\mathbf{a},\mathbf{b})}\}\rangle \\
\texttt{redundant} & \rightarrowtail_{CS} \langle [eq(b,c)], \{\underline{eq(a,b), eq(\mathbf{b},\mathbf{a})}\}\rangle \\
\texttt{transitive} & \rightarrowtail_{CS} \langle [eq(b,c)], \{eq(a,b), eq(b,a), \underline{eq(\mathbf{b},\mathbf{b})}\}\rangle \\
\texttt{reflexive} & \rightarrowtail_{CS} \langle [eq(b,c)], \{\underline{eq(a,b), eq(\mathbf{b},\mathbf{a})}\}\rangle \\
\texttt{transitive} & \rightarrowtail_{CS} \langle [eq(b,c)], \{eq(a,b), eq(b,a), \underline{eq(\mathbf{a},\mathbf{a})}\}\rangle \\
\texttt{reflexive} & \rightarrowtail_{CS} \langle [eq(b,c)], \{eq(a,b), eq(b,a)\}\rangle \\
 & \rightarrowtail_{CS} \langle [], \{eq(a,b), eq(b,a), \underline{eq(\mathbf{b},\mathbf{c})}\}\rangle \\
\texttt{symmetric} & \rightarrowtail_{CS} \langle [], \{eq(a,b), eq(b,a), \underline{eq(b,c), eq(\mathbf{c},\mathbf{b})}\}\rangle \\
\texttt{symmetric} & \rightarrowtail_{CS} \langle [], \{eq(a,b), eq(b,a), \underline{eq(b,c)}, eq(c,b), \underline{eq(\mathbf{b},\mathbf{c})}\}\rangle \\
\texttt{redundant} & \rightarrowtail_{CS} \langle [], \{eq(a,b), eq(b,a), eq(b,c), eq(c,b)\}\rangle \\
 & \vdots \\
 & \rightarrowtail_{CS} \langle [], \{eq(a,b), eq(b,a), eq(b,c), eq(c,b), eq(a,c), eq(c,a)\}\rangle
\end{array}$$

Note that propagation rules are applied at most once to the same sequence of constraints to avoid trivial non-termination.

2.2 Declarative Semantics

Constraints are special predicates of first order logic, their meaning is defined by a constraint theory. A CHR constraint solver program CS has a *logical reading* (meaning) $[\![CS]\!]$. This theory $[\![CS]\!]$ contains the constraint theory for the built-in constraints and one formula for each rule in the program CS.

Let $vars(F)$ denote the set of free variables in formula F, let $\forall F$ denote the universal closure of a formula $\forall vars(F)\ F$, and $\bar{\exists}_V F$ denote the formula $\exists W\ F$ where $W = vars(F) - V$, called the projection of F onto V.

The logical reading of a simpagation rule H_k \ H_r `<=>` G `|` B is:

$$\forall(G \rightarrow (H_k \rightarrow (H_r \leftrightarrow \exists \bar{y}\ B))).$$

where $\bar{y}$ are the variables that appear only in the body B, i.e. $vars(B) - (vars(G) \cup vars(H_k) \cup vars(H_r))$. If any of H_k, H_r, G or B are empty, they are considered *true* in the formula.

Typically the intention of the solver writer is for $[\![CS]\!]$ to approximate some constraint theory $\mathcal{D}$, i.e. $[\![CS]\!]$ covers only the part of $\mathcal{D}$ that is relevant for a particular use. For correctness of the approximation, we require that $[\![CS]\!]$ is a model of a constraint theory $\mathcal{D}$, i.e. $\mathcal{D} \models [\![CS]\!]$ (but typically $[\![CS]\!] \not\models \mathcal{D}$).

Because of the logical reading $[\![CS]\!]$, logical equivalence between successive states is preserved.

Example 2.3 The logical reading $[\![eq]\!]$ of the eq solver of Example 2.1 is the following set of formulas (where we use $\equiv$ for syntactic identity):

reflexive	$x \equiv y \rightarrow (eq(x,y) \leftrightarrow true)$
redundant	$x_1 \equiv x_2 \wedge y_1 \equiv y_2 \rightarrow (eq(x_1,y_1) \rightarrow (eq(x_2,y_2) \leftrightarrow true)$
symmetric	$eq(x,y) \rightarrow eq(y,x)$
transitive	$y_1 \equiv x_2 \rightarrow (eq(x_1,y_1) \wedge eq(x_2,y_2) \rightarrow eq(x_1,y_2))$

Clearly the above rules describe the classical properties of equality.

2.3 Solver Program Properties

A highly useful property of constraint solvers is *confluence*, which ensures that each possible derivation for a goal leads to the same result.

Definition 2.4 [Confluent Solver] A CHR constraint solver *CS* is *confluent* if:

$$\forall C, d, d' : C_1 = solve_d(C) \wedge C_2 = solve_{d'}(C) \Rightarrow \models (\bar{\exists}_{vars(C)} C_1) \leftrightarrow (\bar{\exists}_{vars(C)} C_2)$$

See [1] for a decidable, necessary and sufficient condition for confluence of terminating CHR programs under the general operational semantics.

We typically do not mention the specific derivation d for confluent solvers, if we are not interested in intermediate states and write $C \rightarrowtail^* C_1$ or simply $C_1 = solve(C)$.

Example 2.5 The equality solver *eq* is a confluent solver. For example, for the query `eq(a,a)` a single application of the `reflexive` rule or one application of the `symmetric` and two of the `reflexive` rule both yield an empty constraint store.

A property that combines confluence with the declarative semantics is *canonicity*. It requires that semantically equivalent goals yield the same result.

Definition 2.6 [Canonical Solver] A confluent CHR constraint solver *CS* is *canonical* if:

$$\forall C, C', d, d' : (\llbracket CS \rrbracket \models C \leftrightarrow C') \wedge (C_1 = solve_d(C)) \wedge (C_2 = solve_{d'}(C')) \Rightarrow$$
$$\models (\bar{\exists}_{vars(C \wedge C')} C_1) \leftrightarrow (\bar{\exists}_{vars(C \wedge C')} C_2)$$

In [15] a sufficient, but not necessary, condition for canonical solvers is given. A CHR solver which is confluent, range-restricted and where all simplification rules are single-headed, gives a canonical solver. A CHR solver is *range-restricted* if all variables appearing in the guard and body of the rule appear in the head. In general, it may be non-trivial to show that a solver is canonical.

Example 2.7 The equality solver *eq* is a canonical range-restricted solver. It is obvious that it is range-restricted. Showing it is canonical relies on showing that it returns a store $\{eq(x,y), eq(y,x) \mid$ x and y are connected in the graph created by all the *eq* constraints in the goal $\}$.

3 Basic Implication Checking

Based on the properties of logical implication and conjunction, we can use the following technique to verify whether a constraint c is implied by a conjunction of constraints.

$$D \models C \rightarrow c \quad \Leftrightarrow \quad D \models (C \wedge c) \leftrightarrow C$$

Namely we can use the equivalence of the conjunctions to conclude implication.

In principle, we will verify equivalence in the following way. The solved forms $solve(C)$ and $solve(C \wedge c)$ are computed along some derivations. These solved forms are then projected on the variables of $C \wedge c$ and checked for syntactical equivalence.

This approach is not necessarily complete, but sound, i.e. if the projections of the solved forms are syntactically identical, then c is implied by C.

Theorem 3.1 (Soundness) *For any CHR solver CS:*

$$\forall C, c :\models \bar{\exists}_{vars(C \wedge c)} solve(C) \leftrightarrow \bar{\exists}_{vars(C \wedge c)} solve(C \wedge c) \quad \Rightarrow \quad [\![CS]\!] \models (C \rightarrow c)$$

Proof. We have that $CS \models C \leftrightarrow \bar{\exists}_{vars(C\wedge c)} solve(C)$ and $CS \models C \wedge c \leftrightarrow \bar{\exists}_{vars(C\wedge c)} solve(C)$ from Theorem 4.2 of [7]. Hence if $\models \bar{\exists}_{vars(C\wedge c)} solve(C) \leftrightarrow \bar{\exists}_{vars(C\wedge c)} solve(C \wedge c)$ we have that $CS \models C \leftrightarrow (C \wedge c)$ and hence $CS \models C \rightarrow c$. □

In practical implementations, logical equivalence testing of projected constraint is restricted to syntactic equivalence of multisets ($solve(C) \equiv solve(C \wedge c)$). For range-restricted CHRs the two tests are equivalent since $vars(solve(C)) \subseteq vars(C \wedge c)$ and $vars(solve(C \wedge c)) \subseteq vars(C \wedge c)$.

The straightforward implementation approach for implication checking is to make a copy C' of C, compute both $C \rightarrowtail C''$ and $C' \wedge c \rightarrowtail^* C'''$ and then check equivalence of C'' with C'''. We call this approach the *copy approach*, and it is the implication check currently used by Chameleon [16]. In practice C is already in solved form so $C \equiv C''$.

However, the above copying approach may be quite expensive. C may consist of two parts $C = C_1 \wedge C_2$ such that C_1 is a minimal set of constraints that imply c. By copying C in its entirety C_2 is copied unnecessarily and causes unnecessary overhead in the final equivalence test.

We propose the *trailing* approach for CHR solvers. It only looks at a minimal set of constraints: the conjunction $C \wedge c$ is solved in place and a trail of changes is maintained. Analysis of the trail afterward tells us whether the resulting store is equivalent to the original. If that is the case, the updates to the store may remain. Otherwise, the trail is used to revert to the original situation.

Example 3.2 The following CHR rule conceptually represents the above strategy. Keep in mind the refined semantics with sequential left-to-right execution of the constraints and top to bottom trial of rules.

```
implication @ check_eq(X,Y) <=> eq(X,Y), analyse_trail.
```

The `check_eq/2` constraint represents the implication check. Calling this constraint will either succeed or fail, since `analyse_trail/0` succeeds if the resulting store is equivalent and fails if it is not.

Our trail analysis has to look at the addition and removal of constraints to decide equivalence. Roughly, if any constraint is added or deleted during the implication checking, the resulting store will not be equivalent to the original. More precisely, stores are also equivalent if a constraint is only *temporarily* added or deleted, since addition and deletion of the same constraint cancel each other out.

The following set of CHR rules reflect this approach for `analyse_trail/0`:

```
temporary @ analyse_trail \ added(C), removed(C) <=> true.
addition  @ analyse_trail \ added(C) <=> fail.
removal   @ analyse_trail \ removed(C) <=> fail.
success   @ analyse_trail <=> true.
```

Here `added/1` and `removed/1` represent trail entries of added and deleted constraints.

The code above makes use of the refined semantics [5] to work correctly. A call to `analyse_trail` looks for matching `added/1` and `removed/1` constraints, and removes them using the first rule. If any (unmatched) `added/1` and `removed/1` constraints remain, the second or third rule causes it to fail. Otherwise it reaches the fourth rule which simply succeeds.

In general the original solver is transformed by the rules given in the table below to maintain information about changes.

Entity	New Rule
p	$p(\bar{x})$ `==>` $\mathit{added}(p(\bar{x}))$ add to front of program
$H_k \setminus H_r$ `<=>` G `|` B	$H_k \setminus H_r$ `<=>` G `|` $\mathit{removed}(p_1(\bar{x}_1)), \ldots, \mathit{removed}(p_n(\bar{x}_n)), B$ replace old rule

Example 3.3 The `eq/2` solver is transformed as follows to explicitly generate the necessary `added/1` and `removed/1` constraints.

```
new         @ eq(X,Y) ==> added(eq(X,Y)).
```

```
reflexive   @ eq(X,Y) <=> X == Y | removed(eq(X,Y)).
redundant   @ eq(X1,Y1) \ eq(X2,Y2) <=>
                  X1 == X2, Y1 == Y2 | removed(eq(X2,Y2)).
symmetric   @ eq(X,Y) ==> eq(Y,X).
transitive  @ eq(X1,Y1), eq(X2,Y2) ==> Y1 == X2 | eq(X1,Y2).
```

We want to check whether $eq(a,c)$ is implied by $eq(a,b) \wedge eq(b,c)$. We call the goal `eq(a,b), eq(b,c), check_eq(a,c)`. The first two constraints lead to a store $eq(a,b), eq(b,c), eq(a,c), eq(b,a), eq(c,b), eq(c,a)$ as shown in Example 2.2. The constraint $check\_eq(a,c)$ first adds $added(eq(a,c))$ using the *new* rule, then the *redundant* rule succeeds adding `removed(eq(a,c))`. The call to `analyse_trail` removes both of these using the *temporary* rule, then succeeds using the *success* rule.

The transformed solver program above has the disadvantage that it always trails. The following set of rules only enable explicit trailing during implication checking. These rules require `trail_off` to be in the constraint store initially.

```
implication   @ check_eq(X,Y) <=> enable_trail,
                                        eq(X,Y), analyse_trail,
                                  disable_trail.
enable        @ trail_off, enable_trail <=> trail_on.
disable       @ trail_on, disable_trail <=> trail_off.
filter_add    @ trail_off \ added(C) <=> true.
filter_remove @ trail_off \ removed(C) <=> true.
```

Our method of implication checking is complete for canonical range-restricted CHR-only solvers.

Theorem 3.4 (Completeness) *If CS is a canonical range-restricted solver, then implication checking is complete. That is*

$$\forall C, c : ([\![CS]\!] \models C \Rightarrow c) \Rightarrow solve(C) \equiv solve(C \wedge c)$$

Proof. From Definition 2.6 we have that

$$\forall C, c : ([\![CS]\!] \models C \leftrightarrow (C \wedge c) \Rightarrow \bar{\exists}_{vars(C \wedge c)} solve(C) \leftrightarrow \bar{\exists}_{vars(C \wedge c)} solve(C \wedge c)$$

Since CS is range-restricted $vars(solve(C)) \subseteq vars(C \wedge c)$ and $vars(solve(C \wedge c)) \subseteq vars(C \wedge c)$. Hence $solve(C) \equiv solve(C \wedge c)$. □

Note that in the special case that $C \wedge c$ fails, c is not implied by C as $C \wedge c$ is not satisfiable. This case is correctly covered by our approach.

A constraint solver need not be canonical for our implication checking to be complete. In Section 5 we will show that our method is also complete for several non-canonical, even non-confluent, CHR solvers.

4 Implication Checking for Modular Solver Hierarchies

In this section we extend the implication checking technique of the previous section to modular CHR solver hierarchies. Constraint solvers may be nested hierarchically: one solver depends on some solvers that on their turn depend on other solvers. We say a solver *depends* on another solver if it uses constraints that are defined in the other solver. In particular a CHR solver can uses constraints of other solvers in guards and bodies of CHR rules.

Previously it was only possible to use constraints of builtin solvers in guards [4]. Here we show how to construct CHR solver hierarchies where CHR constraints can be used in guards. In a CHR solver hierarchy the dependency graph is acyclic. We will use parent solver and child solver to refer to one solver that depends on the other.

A *modular* CHR solver is a CHR solver that can be compiled using the interface of its dependencies only. In particular, no knowledge of dependents is required.

Example 4.1 The following less-than-or-equal-to solver *leq* depends on the *eq* solver:

```
leq_new           @ leq(X,Y) <=> check_eq(X,Y) | true.
leq_antisymmetric @ leq(X1,Y1), leq(X2,Y2) <=>
                    check_eq(X1,Y2), check_eq(X2,Y1) | eq(X1,Y1).
leq_redundant     @ leq(X1,Y1) \ leq(X2,Y2) <=>
                    check_eq(X1,X2), check_eq(Y1,Y2) | true.
leq_transitive    @ leq(X1,Y1), leq(X2,Y2) ==>
                    check_eq(Y1,X2) | leq(X1,Y1).
```

This *leq* solver depends on the *eq* solver in two ways. Firstly, it calls `eq/2` constraints in the body of the `leq_antisymmetric` rule. Secondly, it also uses the `check_eq/2` implication check in the guard of all its rules. As both the constraint and the implication check can easily be exported from the *eq* solver this does not violate modularity.

There is a part of the operational semantics of the guard that we have not addressed yet. We call an *event* the addition of a constraint to the child solver or one of the solvers it depends on, such that the guard may be satisfied. A CHR rule application may not succeed immediately because a guard is not satisfied, but an event may cause it to be satisfied at a later point.

The semantics of CHR require that CHR constraints of the parent solver are re-activated in case of an event that now satisfies a previously unsatisfied guard. In practice, CHR implementations overestimate the impact of events, i.e. re-activate more than necessary. Typically for builtin solvers, relevant events are provided in the solver interface by the solver programmer together with a mechanism to notify interested parties.

The following rules describe the necessary operations for such a mechanism of events and notifications for the *eq* solver:

```
new_event @ eq(X,Y) ==> touched(X), touched(Y).
trigger   @ touched(X), delayed(X,Goal,ID) ==> call(Goal).
end_event @ touched(X) <=> true.
kill_goal @ kill(ID) \ delayed(X,Goal,ID) <=> true.
kill_end  @ kill(ID) <=> true.
```

The *eq* solver provides a `touched(X)` event in its interface, without knowing anything about particular uses. The `new_event` rule generates the `touched` event for every variable involved in a new `eq/2` constraint. Users of the interface, such as the *leq* solver will be notified of these events by calling the `delayed/3` constraint. This constraint supplies a callback goal, that is called when the appropriate `touched/1` event fires and allows the notified party to take due action. The `kill/1` constraint allows for the removal of one or more delayed callbacks, based on an identifier and allows the notified party to no longer receive any events.

The following pseudo-code shows how the *leq* solver subscribes itself to `touched` events. It is pseudo-code because it accesses some internals of the CHR implementation.

```
listen @ leq(X,Y) # CID ==> new_delay_id(ID),
                            delay(X,reactivate(CID),ID),
                            delay(Y,reactivate(CID),ID),
                            listening(CID,ID).
```

Here `CID` is an internal identifier of the CHR constraint. This pseudo-rule is executed when the `leq(X,Y)` constraint is first activated. The call to `new_delay_id/1` generates a new notification identifier. With the two calls to `delay` the $\leq$ solver will be notified of the relevant events. Upon notification the internal goal `reactivate(CID)` is called which reactivates the corresponding constraint. The call to `listening` internally associates the notification identifier with the corresponding `leq/2` constraint. When the `leq/2` constraint with identifier `CID` is removed, internally the `kill/1` constraint is called on all associated notification identifiers. This avoids reactivation of removed constraints.

However, several modifications to the implication checking are now necessary to the original scheme, to accommodate both the hierarchy and the modularity. In the following we explain how to do trailing for multiple CHR solvers, how to distinguish between trails of recursively called implication checks and how implication checking should interact with the event mechanism.

Trailing Interface: Firstly, because of the hierarchy, during an implication check on a parent solver, constraints in the child solver may be added and deleted. Hence, the parent solver trail mechanism has to recursively rely on the child solver trail mechanism. The child solver has to export the necessary trail operations for this.

Example 4.2 The following set of rules encode the trailing dependency of the `leq/2` solver on the `eq/2` solver:

```
rec_analysis  @ leq_analyse_trail ==> eq_analyse_trail.
rec_enable    @ leq_enable_trail  ==> eq_enable_trail.
rec_disable   @ leq_disable_trail ==> eq_disable_trail.
```

Implication Strata: Secondly, because of the hierarchy, implication checking may be recursive as well. For example, an implication check of a `leq/2` constraint may require the implication check of a `eq/2` constraint. Our *shallow* trailing approach does not cover this any more. Indeed, it does not distinguish between those `eq/2` constraints added and deleted during the recursive `eq/2` implication check and those during the top level `leq/2` implication check. A more involved trailing mechanism is needed.

Our solution is to associate with each implication stratum (i.e. level of implication check nesting) a stratum identifier. The top level which is not inside any implication check has stratum identifier 0, an implication check called from top level has identifier -1, etc.

Every constraint is labeled with the stratum it is called in. For example, `eq(X,Y)` becomes `eq(X,Y,S)` if it is called in stratum `S`.

Constraints called in the top-level query are assigned stratum 0. Constraints called in the body of a rule inherit the lowest stratum of any constraints in the head and the implication checking lowers the stratum by one.

Example 4.3 For example, the `leq/2` solver is transformed as follows:

```
leq_new           @ leq(X,Y,S) <=> check_eq(X,Y,S-1) | true.
leq_antisymmetric @ leq(X1,Y1,S1), leq(X2,Y2,S2) <=>
     check_eq(X1,Y2,min(S1,S2)-1), check_eq(X2,Y1,min(S1,S2)-1)
                    | eq(X1,Y1,min(S1,S2)).
leq_redundant     @ leq(X1,Y1,S1) \ leq(X2,Y2,S2) <=>
     check_eq(X1,X2,min(S1,S2)-1), check_eq(Y1,Y2,min(S1,S2)-1)
                    | true.
leq_transitive    @ leq(X1,Y1,S1), leq(X2,Y2,S2) ==>
     check_eq(Y1,X2,min(S1,S2)-1) | leq(X1,Y1,min(S1,S2)).
```

Now it is possible for the implication trailing operations to work on a single stratum by looking at the stratum identifiers: all the related constraints are extended with their stratum's identifier.

However, the implication checking is weakened, if the trailing operations are confined within a stratum. The reason is the `temporary` rule:

```
temporary @ analyse_trail(S) \ added(C,S), removed(C,S) <=> true.
```

This rule only cancels out additions and deletions in the same stratum. What is no longer canceled out, is a constraint added in a higher stratum that is removed in a lower stratum and re-added in that lower stratum.

It is possible to recapture this possibility as follows. With every deletion both the stratum of the deleted constraint and the lowest stratum of any of the head constraints is recorded. The latter is the *cause* of the removal. For example, the leq2 rule then looks like:

```
leq2 @ leq(X1,Y1,S1), leq(X2,Y2,S2) <=>
        check_eq(X1,Y2,min(S1,S2)-1), check_eq(X2,Y1,min(S1,S2)-1)
        | removed(leq(X1,Y1),S1,min(S1,S2)),
          removed(leq(X2,Y2),S2,min(S1,S2)),
          eq(X1,Y1,min(S1,S2)).
```

The following rules deal with this new removed/3 constraint.

```
temporary @ analyse_trail(S) \ added(leq(X,Y),S),
                                removed(leq(X,Y),S,S)
                              <=> true.
promotion @ analyse_trail(S) \ added(leq(X,Y),S), leq(X,Y,S),
                                removed(leq(X,Y),Sr,S)
                              <=> S < Sr | leq(X,Y,Sr).
```

The temporary rule still cancels out addition and deletion within the same stratum, but the promotion rule promotes a new constraint to the stratum of the previously deleted constraint. In this way the full power of the basic implication checking is retained for solver hierarchies.

The definition of check_eq/3 becomes the following. We can get rid of explicit trail enabling and disabling now that we have the implication strata: stratum 0 never requires trailing and the other strata always do.

```
toplevel_add @ added(_,0)      <=> true.
toplevel_rem @ removed(_,_,0)  <=> true.

implication  @ check_eq(X,Y,S) <=> eq(X,Y,S),
                                   eq_analyse_trail(S).
```

Inter-stratum Events: During an implication check which takes place in the child solver an event may be fired waking some parent solver constraints that cause some parent solver constraints to be added or deleted in a higher stratum.

However, it is not necessary for these events to travel across strata. An implication check can safely be resolved without propagating any information to the parent solver in the higher stratum: as a child solver does not depend on the parent solver the outcome of an implication check on the child solver should not require interaction with the parent solver.

The other way around, a higher stratum will never generate any event in the presence of a lower stratum, since it is temporarily suspended while execution goes on in the lower stratum and only disappears after the implication check in the lower stratum has finished and thus the lower stratum is gone altogether.

Hence, is safe and cheaper for events to only trigger callbacks within the same stratum. The following modified event code reflects this.

```
new_event @ eq(X,Y,S) ==> touched(X,S), touched(Y,S).
trigger   @ touched(X,S), delayed(X,Goal,S,ID) ==> call(Goal).
end_event @ touched(X,S) <=> true.

listen    @ leq(X,Y,S) # CID ==> new_delay_id(ID),
                                 delay(X,reactivate(CID),S,ID),
                                 delay(Y,reactivate(CID),S,ID),
                                 listening(CID,ID).
```

5 Case Studies: Non-Canonical Solvers

We have shown in Section 3 that our CHR implication checking is complete for canonical solvers. In this section we investigate the completeness for some classical, non-canonical solvers.

It will turn out that the implication checking is still complete in many cases, or can be made complete with a little customization in particular cases.

5.1 Naive Union-Find Equality Solver

In [13] a CHR implementation of the naive union-find algorithm is presented (see Appendix A for the source code). The `union/2` constraint in that implementation may serve as an equality constraint.

The naive union-find represents equal variables as nodes in the same tree. Any tree with the same variables in it represents the equality of its elements. There is not one preferred, canonical form. For this reason it is even non-confluent: the order of the `union/2` constraints, decides the shape of the tree.

If two variables are unioned that are already equal, their common tree is not modified, nor are any other constraints deleted or added. However, if two variables are not yet equal, a union will merge their trees into one.

Hence, our implication check is complete for this union-find equality solver.

5.2 Optimal Union-Find Equality Solver

Next to the naive algorithm also a CHR implementation of an optimal union-find algorithm is given in [12] (see Appendix B for the source code). This algorithm combines path compression with union-by-rank.

Again, when two variables are not equal, their respective trees are merged (by-rank) and this is detected by our implication method. However, in case the variables are already equal, path compression may still modify the tree by shortening paths from nodes to the root. Because the compressed tree is not syntactically identical to the initial tree, our implication method will reject it.

Nevertheless it is possible to customize the `trail_analysis/0` rules to overcome this problem and safely allow path compression, while rejecting truly new equalities. Namely, instead of these general rules:

```
addition  @ analyse_trail(S) \ added(C,S) <=> fail.
removal   @ analyse_trail(S) \ removed(C,_,S) <=> fail.
```

only the detection of the removal of a `root` constraint is required to detect the linking of two trees:

```
removal   @ analyse_trail(S) \ removed(root(_,_),_,S) <=> fail.
cleanup1  @ analyse_trail(S) \ added(_,S) <=> true.
cleanup2  @ analyse_trail(S) \ removed(_,_,S) <=> true.
cleanup3  @ analyse_trail(S) \ '~>'(X,Y,S) <=> '~>'(X,Y,S+1).
```

Since these rules do not consider path compression as possible non-equivalence of trees. Indeed, they even will not undo the path compression after the implication check, but promote newly created edges to the higher stratum. Hence the compacter tree representation is retained after a succeeding implication check, making future operations cheaper.

6 Experimental Evaluation

We compare our trailing approach with the naive copy approach to validate it. For this reason we consider a particular benchmark for the *eq* solver. $n-1$ constraints $\texttt{eq(V}_i\texttt{,V}_{i+1}\texttt{)}$ are imposed for $1 \leq i < n$. This conjunction of equality constraints we call C. The constraint we test for, c, is $\texttt{eq(V}_1\texttt{,V}_n\texttt{)}$ in one case and $\texttt{eq(V}_1\texttt{,V}_{n+1}\texttt{)}$ in the other. The former test succeeds and the latter fails.

Table 1 lists the experimental results in seconds obtained for this benchmark with $n = 20$ using the K.U.Leuven CHR system in SWI-Prolog on an Intel Pentium 4 2.0GHz with 512MB of RAM. The total time to perform implication checking for the copy approach is equal to the sum of the times for $solve(C)$ and $solve(C \wedge c)$[1]. With our trailing approach, the time to compute $C \implies c$ corresponds with the time for $solve(C \wedge c)$. While there is about 16% overhead for ordinary use ($solve(C)$), the trailing approach is clearly superior to the copy approach for implication testing: it is almost twice as fast for our benchmark. The succeeding test performs a little better than the failing one.

The table also lists the results for a similar benchmark using the naive union-find program. The trailing version has been specialized to not trail additions and removals of constraints that are never stored. The results are similar as for the *eq* solver.

7 Conclusion

In this paper we have presented a new approach for automatic implication checking in CHR solvers. We have established the soundness of our trailing approach as well as its completeness for the class of canonical CHR solvers. In

[1] The time to compare the constraint stores is negligible for this benchmark.

	Approach	*solve*(C)	*solve*$(C \wedge c)$	$C \implies c$	
$c =$ `eq(V`$_1$`,V`$_n$`)`	copy	4.34	4.42	9.16	100.0%
	trailing	5.02	5.07	5.07	55.2%
$c =$ `eq(V`$_1$`,V`$_{n+1}$`)`	copy	4.34	5.61	9.95	100.0%
	trailing	5.02	6.53	6.53	65.6%
$c =$ `union(V`$_1$`,V`$_n$`)`	copy	2.78	2.80	5.58	100.0%
	trailing	3.75	3.77	3.77	67.6%
$c =$ `union(V`$_1$`,V`$_{n+1}$`)`	copy	2.78	2.79	5.57	100.0%
	trailing	3.75	3.78	3.78	67.9%

Table 1
Experimental Results

addition we have studied the completeness for several existing CHR solvers. Experimental evaluation supports our claim that the trailing approach is more efficient than a naive copying approach. In addition we have extended our trailing approach to CHR solver hierarchies. We can now use CHR constraints of one solver in the guards of rules of another solver.

Acknowledgments

We would like to thank Marc Meister for commenting on a preliminary version of this paper. Part of this research was conducted while Tom Schrijvers was visiting the University of Melbourne in July 2004 and the Universität Ulm in November 2004.

References

[1] Abdennadher, S., *Operational Semantics and Confluence of Constraint Propagation Rules*, in: G. Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, 1997, pp. 252–266.

[2] Carlsson, M., G. Ottosson and B. Carlson, *An Open-Ended Finite Domain Constraint Solver*, in: H. Glaser, P. H. Hartel and H. Kuchen, editors, *PLILP'97: Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs*, Lecture Notes in Computer Science **1292** (1997), pp. 191–206.

[3] Duck, G. J., M. Garcia de la Banda and P. J. Stuckey, *Compiling Ask Constraints*, in: B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming*, LNCS (2004), pp. 105–119.

[4] Duck, G. J., 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* (2003), pp. 79–90.

[5] Duck, G. J., P. J. Stuckey, M. Garcia de la Banda and C. Holzbaur, *The Refined Operational Semantics of Constraint Handling Rules*, in: *20th International Conference on Logic Programming (ICLP'04)*, Saint-Malo, France, 2004, pp. 90–104.

[6] Frühwirth, T., *Entailment Simplification and Constraint Constructors for User-Defined Constraints*, in: *3rd Workshop on Constraint Logic Programming (WCLP 93)*, Marseille, France, 1993.

[7] Frühwirth, T., *Theory and Practice of Constraint Handling Rules*, Journal of Logic Programming **37** (1998), pp. 95–138.

[8] Holzbaur, C. and T. Frühwirth, *Compiling Constraint Handling Rules into Prolog with Attributed Variables*, in: G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, number 1702 in LNCS (1999), pp. 117–133.

[9] Holzbaur, C. and T. Frühwirth, *Constraint Handling Rules, Special Issue*, Journal of Applied Artificial Intelligence **14** (2000).

[10] Holzbaur, C., M. García de la Banda, P. J. Stuckey and G. J. Duck, *Optimizing Compilation of Constraint Handling Rules in HAL*, Special Issue of Theory and Practice of Logic Programming on Constraint Handling Rules (2005), to appear.

[11] Schrijvers, T. 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*, 2004, pp. 1–5.

[12] Schrijvers, T. and T. Frühwirth, *Implementing and Analysing Union-Find in CHR*, Report CW 389, K.U.Leuven, Department of Computer Science (2004).

[13] Schrijvers, T. and T. Frühwirth, *Optimal Union-Find in Constraint Handling Rules*, Theory and Practice of Logic Programming (2005), to appear.

[14] Schulte, C., *Programming Deep Concurrent Constraint Combinators*, in: E. Pontelli and V. S. Costa, editors, *PADL'00: 2nd International Workhop of Practical Aspects of Declarative Languages*, Lecture Notes in Computer Science **1753** (2000), pp. 215–229.

[15] Stuckey, P. J. and M. Sulzmann, *A Theory of Overloading*, in: *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming* (2002), pp. 167–178.

[16] Stuckey, P. J., M. Sulzmann and J. Wazny, *The Chameleon System*, in: T. Frühwirth and M. Meister, editors, *First workshop on constraint handling rules: selected contributions*, 2004, pp. 13–32.

A Source Code: Naive Union-Find

Naive Union-Find

```
make     @ make(X) <=> root(X).

union    @ union(X,Y) <=> find(X,A), find(Y,B), link(A,B).

findNode @ X ~> PX \ find(X,R) <=> find(PX,R).
findRoot @ root(X) \ find(X,R) <=> R=X.

linkEq   @ link(X,X) <=> true.
link     @ link(X,Y), root(X), root(Y) <=> Y ~> X, root(X).
```

B Source Code: Optimal Union-Find

ufd_rank

```
make      @ make(X) <=> root(X,0).

findNode  @ X ~> PX , find(X,R) <=> find(PX,R), X ~> R.
findRoot  @ root(X,_) \ find(X,R) <=> R=X.

linkEq    @ link(X,X) <=> true.
linkLeft  @ link(X,Y), root(X,RX) root(Y,RY) <=> RX >= RY |
                Y ~> X, NRX is max(RX,RY+1), root(X,NRX).
linkRight @ link(X,Y), root(Y,RY), root(X,RX) <=> RY >= RX |
                X ~> Y, NRY is max(RY,RX+1), root(Y,NRY).
```