INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67, D–80538 München

Ludwig
Maximilians–
Universität
München

Confluent Simplification Rules

Slim Abdennadher, Thom Frühwirth, Holger Meuss

http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1996-7, Januar 1996

Confluent Simplification Rules

Slim Abdennadher[1], Thom Frühwirth[2], Holger Meuss[1]

[1]Computer Science Department, University of Munich
Wagmüllerstr. 23, 80538 Munich, Germany
{Slim.Abdennadher,Holger.Meuss}@informatik.uni-muenchen.de

[2]ECRC, Arabellastr. 17, Munich, Germany
Thom.Fruehwirth@ecrc.de

Abstract

We introduce the notion of confluence for Constraint Handling Rules (CHR), a powerful language for writing constraint solvers. With CHR one simplifies and solves constraints by applying rules. Confluence guarantees that a CHR program will always compute the same result for a given set of constraints independent of which rules are applied. We give a decidable, sufficient and necessary syntactic condition for confluence.

Confluence turns out to be an essential syntactical property of CHR programs for two reasons. First, confluence implies correctness (as will be shown in this paper). In a correct CHR program, application of CHR rules preserves logical equivalence of the simplified constraints. Secondly, even when the program is already correct, confluence is highly desirable. Otherwise, given some constraints, one computation may detect their inconsistency while another one may just simplify them into a still complex constraint.

As a side-effect, the paper also gives soundness and completeness results for CHR programs. Due to their special nature, and in particular correctness, these theorems are stronger than what holds for the related families of (concurrent) constraint programming languages.

Keywords: constraint reasoning, semantics of programming languages, committed-choice languages, confluence and determinacy.

1 Introduction

Constraint Handling Rules (CHR) [Frü95] have been designed as a special-purpose language for writing constraint solvers. A constraint solver stores and simplifies incoming constraints. CHR is essentially a committed-choice language consisting of guarded rules that rewrite constraints into simpler ones until they are solved.

In contrast to the family of the general-purpose concurrent constraint languages

(CC) [Sar93] and the ALPS[1] [Mah87] framework, CHR allow "multiple heads", i.e. conjunctions of atoms in the head of a rule. Multiple heads are a feature that is essential in solving conjunctions of constraints. With single-headed CHR rules alone, unsatisfiability of constraints could not always be detected (e.g `X<Y,Y<X`) and global constraint satisfaction could not be achieved.

Nondeterminacy in CHR arises when two or more rules can fire. It is obviously desirable that the result of a computation in a solver will always be the same, semantically and syntactically, no matter in which CHR rules are applied. This property of constraint solvers will be called confluence and investigated in this paper.

We will introduce a decidable, sufficient and necessary syntactic condition for confluence. This condition adopts the notion of critical pairs as known from term rewrite systems [DOS88, KK91, Pla93]. Monotonicity of constraint store updates, an inherent property of constraint logic programming languages, plays a central role in proving that joinability of critical pairs is sufficient for local confluence.

Confluence turns out to be important with regard to both theoretical and practical aspects: We show that confluence implies correctness of a program. By correctness we mean that the declarative semantic of a CHR program is a consistent theory. Unlike CC programs, CHR programs can be given a declarative semantics since they are only concerned with defining constraints (i.e. first order predicates), not procedures in their generality. Furthermore we show how to strengthen the declarative reading of a CHR program if it is confluent. A practical application of our definition of confluence lies in program analysis, where we can identify non–confluent parts of CHR programs by examining the critical pairs. Programs with non–confluent parts essentially represent an ill-defined constraint solving algorithm.

Our work extends previous approaches to the notion of determinacy in the field of CC languages: Maher investigates in [Mah87] a class of flat committed choice logic languages (ALPS). He defines the class of deterministic ALPS programs as those programs whose guards are mutually exclusive. The class of deterministic ALPS programs is less expressive than confluent CHR programs. Saraswat defines for the CC framework a similar notion of determinacy [Sar93], which is also more restrictive than confluence. We also give two reasons, why CHR cannot be made deterministic in general.

Our approach is orthogonal to the work in program analysis in [MO95] and [FGMP95], where a different, less rigid notion of confluence is defined: A CC program is confluent, if different process schedulings (i.e. different orderings of decisions at nondeterministic choice points) give rise to the same set of possible outcomes. The idea of [MO95] is to introduce a non-standard semantics, which is confluent for all CC programs.

The paper is organized as follows. The next section introduces the syntax of constraint handling rules, their declarative and operational semantics. Then this section contributes to the relationship between the declarative and operational semantics of CHR programs by giving soundness and completeness results. Section 3 presents the notion of confluence for CHR. In section 4 we show that

[1] Saraswat showed in [Sar93], that ALPS can be recognized as a subset of cc($\downarrow, \rightarrow$)

confluence implies logical correctness of a program. This leads to a stronger completeness and soundness result for finite failed computation. Finally, we conclude with a summary and directions for future work.

2 Syntax and Semantics of CHR

We assume some familiarity with (concurrent) constraint programming (CCP) [JL87, JM94, SRP91, Sar93, Sha89]. There is a distinguished class of predicates, the *constraints*. We assume, that there is a built-in constraint solver that solves, checks and simplifies built-in (predefined) constraints. On the other hand, the user-defined constraints are those defined by a CHR program. This implies, that we have two disjoint sets of constraint symbols for the built–in and the user–defined constraints.

As a special purpose language, CHR usually extend a host language such as Prolog or Lisp with (more) constraint solving capabilities. This also means, that auxiliary computations in CHR programs can be performed in the host language. Without loss of generality, to keep this paper self-contained, we will not address host language issues here. We also restrict ourselves to the main kind of CHR rule.

Definition 2.1 A CHR *program* is a finite set of simplification rules[2]. A *simplification rule* is of the form

$$H_1, \ldots, H_i \Leftrightarrow G_1, \ldots, G_j \mid B_1, \ldots, B_k.$$

where the multi–head $H_1, \ldots, H_i$ is a conjunction[3] of user-defined constraints and the guard $G_1, \ldots, G_j$ is a conjunction of built-in constraints and the body $B_1, \ldots, B_k$ is a conjunction of built-in and user-defined constraints called goals.

2.1 Declarative Semantics

Unlike CC programs, CHR programs can be a given a declarative semantics since they are only concerned with defining constraints (i.e. first order predicates), not procedures in their generality.

Declaratively, a simplification rule

$$H_1, \ldots, H_i \Leftrightarrow G_1, \ldots, G_j \mid B_1, \ldots, B_k.$$

is a logical equivalence provided the guard is true in the current context

$$\forall \bar{x}\ (\exists \bar{y}\ (G_1 \wedge \ldots \wedge G_j)) \rightarrow (H_1 \wedge \ldots \wedge H_n \leftrightarrow \exists \bar{z}\ (B_1 \wedge \ldots \wedge B_k)),$$

where $\bar{x}$[4] are the variables occuring in $H_1, \ldots, H_n$ and $\bar{y}, \bar{z}$ are the other variables occuring in $G_1, \ldots, G_j$ and $B_1, \ldots B_k$ respectively.

[2] There are two other kinds of rules [BFL+94], which are not treated here.

[3] For conjunctions in rules we use "," instead of "$\wedge$".

[4] we use $\bar{x}$ as an abbreviation for a sequence of variables

The declarative interpretation of a CHR program P is given by the set $\mathcal{P}$ of logical equivalences and a consistent built–in theory CT which determines the meaning of the built–in constraints appearing in the program. The constraint theory CT specifies among other things the ACI properties of the logical conjunction $\wedge$ in the built-in constraint store, the properties of the equality constraints $\doteq$ (Clarks axiomatization) and the properties of the basic constraints *true* and *false*.

Definition 2.2 A CHR program P is *correct* iff $\mathcal{P} \cup CT$ is consistent.

2.2 Operational Semantics of CHR

We define the operational semantics as a transition system.

2.2.1 States

Definition 2.3 A *state* is a triple

$$<C_U, C_B, \mathcal{V}>.$$

C_U is a conjunction of both user-defined and built-in constraints that remains to be solved. C_B is a conjunction of built-in constraints accumulated up to this point of execution. $\mathcal{V}$ is an ordered set of variables.

Definition 2.4 A variable X in a state $<C_U, C_B, \mathcal{V}>$ is called *global*, if it appears in $\mathcal{V}$. It is called *local* otherwise.

Definition 2.5 The pair (C_1, C_2) (C_1 and C_2 are conjunctions of constraints) is called *enclosed* by the ordered set $\mathcal{V}$ iff all variables shared by C_1 and C_2 are contained in $\mathcal{V}$.

We can attribute to each state $<C_U, C_B, \mathcal{V}>$ the formula

$$\exists Y_1, \ldots, Y_m \; C_U \wedge C_B$$

as a logical meaning, where $Y_1, \ldots, Y_m$ are the local variables in C_U and C_B. Note that the global variables remain unbound in the formula.

2.2.2 Update

We define now the basic operation of the built–in constraint solver: The main task of *update* is transforming a state into a logically equivalent state with a normalized built–in constraint store. update performs the following tasks:

- normalize the built–in constraint store according to CT
- propagate equality constraints through the state
- remove redundant equality constraints where one side is a local variable.

Definition 2.6 update normalizes a state by performing the following operations in sequence:

1. update produces a unique representation of the built-in constraint store according to the theory CT.
2. Equality constraints of the form $X \doteq t$ receive a special treatment: occurrences of X in all constraints (except the equality itself) in the built-in constraint store and goal store are replaced by t.
3. All equality constraints of the form $X \doteq t$ or $Y \doteq X$ are removed, if X is local. These equality constraints will be called *local*. This reflects the validity of formulas $(\exists X\ X \doteq a)$, which follows from the axioms in CT (see example 2.1).

Example 2.1

$$\mathrm{update}(<p(Y) \wedge q(Z), Y \doteq f(X) \wedge Z \doteq a, [Y]>) = <p(f(X)) \wedge q(a), Y \doteq f(X), [Y]>$$

Under an enclosement condition update is compatible with addition of constraints. This result is given by the following lemma, which is proven by contradiction.

Lemma 2.1 If C is a conjunction of built–in constraints and (C, C_B) is enclosed by $\mathcal{V}$ and $\mathrm{update}(<C_U, C_B, \mathcal{V}>) = <C'_U, C'_B, \mathcal{V}>$ then

$$\mathrm{update}(<C_U, C_B \wedge C, \mathcal{V}>) = \mathrm{update}(<C'_U, C'_B \wedge C, \mathcal{V}>).$$

The enclosement condition in the lemma above reflects the sensitivity of update with respect to local variables. It guarantees that equality constraints involving variables appearing in the added constraint C are not removed due to locality. If the condition is violated, the claim is false:

Example 2.2

$$\mathrm{update}(<\mathrm{true}, X \doteq 2, []>) = <\mathrm{true}, \mathrm{true}, []>,$$

adding the built-in constraint $X \doteq 1$ on both sides results for the left side in:

$$\mathrm{update}(<\mathrm{true}, X \doteq 2 \wedge X \doteq 1, []>) = <\mathrm{true}, \mathrm{false}, []>$$

but for the right side in:

$$\mathrm{update}(<\mathrm{true}, \mathrm{true} \wedge X \doteq 1, []>) = <\mathrm{true}, \mathrm{true}, []>$$

Definition 2.7 *Entailment* $(\rightarrow_o)$ tests whether a given conjunction of built–in constraints is implied by another conjunction of built–in constraints in the context of a state and is defined as follows:

$$\begin{array}{rcl} <C_{U1}, C_{B1}, \mathcal{V}> & \rightarrow_o & <C_{U2}, C_{B2}, \mathcal{V}> \quad \text{iff} \\ <C'_{U1}, C'_{B1}, \mathcal{V}> & = & \mathrm{update}(<C'_{U2}, C'_{B1} \wedge C'_{B2}, \mathcal{V}>). \end{array}$$

where $\mathrm{update}(<C_{U1}, C_{B1}, \mathcal{V}>) = <C'_{U1}, C'_{B1}, \mathcal{V}>$ and $\mathrm{update}(<C_{U2}, C_{B2}, \mathcal{V}>) = <C'_{U2}, C'_{B2}, \mathcal{V}>$.

2.2.3 Computation Steps

Given a CHR program P we define the transition relation $\mapsto_P$ by introducing two kinds of *computation steps*:

> **Solve** $<C \wedge C_U, C_B, \mathcal{V}> \;\; \mapsto_P \;\; \text{update}(<C_U, C \wedge C_B, \mathcal{V}>)$
> if C is a built-in constraint.

The built–in constraint solver updates the state after adding the built–in constraint C to the built–in store C_B.

> **Simplify** $<H' \wedge C_U, C_B, \mathcal{V}> \;\; \mapsto_P \;\; \text{update}(<C_U \wedge B, H \dot{=} H' \wedge C_B, \mathcal{V}>)$
> if $(H \Leftrightarrow G \mid B)$ is a variant with fresh variables of a rule in P and
> $<H', C_B, \mathcal{V}> \rightarrow_o <H', H' \dot{=} H \wedge G, \mathcal{V}>$.

To simplify user-defined atoms means to apply a simplification rule on these atoms. This can be done if the atoms match with the head atoms of the rule and the guard is entailed by the built-in constraint store. The atoms occuring in the body of the rule are added to the goal constraint store.

Notation. By $c(t_1, \ldots, t_n) \dot{=} c(s_1, \ldots, s_n)$ we mean $t_1 \dot{=} s_1 \wedge \ldots \wedge t_n \dot{=} s_n$, if c is a user-defined constraint. By $p_1 \wedge \ldots \wedge p_n \dot{=} q_1 \wedge \ldots \wedge q_n$ we mean $p_1 \dot{=} q_1 \wedge \ldots \wedge p_n \dot{=} q_n$.

Definition 2.8 $S \mapsto_P^* S'$ holds iff

$$S = S' \text{ or } S = \text{update}(S') \text{ or } S \mapsto_P S_1 \mapsto_P \ldots \mapsto_P S_n \mapsto_P S' \quad (n \geq 0).$$

We will write $\mapsto$ instead of $\mapsto_P$ and $\mapsto^*$ instead of $\mapsto_P^*$, if the program P is fixed.

Lemma 2.2 Update has no influence on application of rules, i.e.

$$S \mapsto S' \text{ implies update}(S) \mapsto S'.$$

The *initial state* consists of a goal G, an empty built-in constraint store and the list $\mathcal{V}$ of the variables occuring in G,

$$<G, \text{true}, \mathcal{V}>.$$

A computation state is a *final state* if

- its built-in constraint store is false, then it is called *failed*;
- no computation step can be applied and its built-in constraint store is not false. Then it is called *successful*.

Definition 2.9 A *computation* of a goal G is a sequence $S_0, S_1, \ldots$ of states with $S_i \mapsto S_{i+1}$ beginning with the the initial state $S_0 = <G, \text{true}, \mathcal{V}>$ and ending in a final state or diverging. A finite computation is *successful* if the final state is successful. It is *failed* otherwise.

Definition 2.10 A *computable constraint* C of G is the conjunction $\exists\bar{x}\ C_U \wedge C_B$, where C_U and C_B occur in a state $<C_U, C_B, \mathcal{V}>$, which appears in a computation of G. $\bar{x}$ are the local variables.
A *final constraint* C is the conjunction $\exists\bar{x}\ C_U \wedge C_B$, where C_U and C_B occur in a final state $<C_U, C_B, \mathcal{V}>$.

2.2.4 Equivalence and Monotonicity

The following definition reflects the AC1 properties of the goal store and the fact that all states with an inconsistent built–in constraint store are identified.

Definition 2.11 We identify states according to the equivalence relation $\cong$:
$<C_U, C_B, \mathcal{V}> \cong <C'_U, C_B, \mathcal{V}>$ iff C_U can be transformed to C'_U using the AC1 properties of the conjunction $\wedge$, or C_B is *false*.

We have to ensure that the equivalence $\cong$ is well-defined, i.e. that it is compatible with the operations we perform on states. We have six different operations working on states, 1-3 are explicitly used for computation steps, whereas 4-6 occur only in the proof for the theorem on local confluence:

1. **Solve**
2. **Simplify**
3. update
4. add a constraint to the goal store or built-in constraint store
5. form a variant
6. replace the global variable store by another ordered set of variables

It is easy to see that all these operations are congruent with the relation $\cong$, i.e. the following holds for each instance o of an operation:

$$S_1 \cong S_2 \text{ implies } o(S_1) \cong o(S_2)$$

Therefore we can reason about states modulo $\cong$.

The next definition defines the notion of monotonicity, which guarantees that addition of new built-in constraints does not inhibit entailment (and hence the application of **Simplify**):

Definition 2.12 A built–in constraint solver is said to be monotonic iff the following holds:

$$<C_{U1}, C_B, \mathcal{V}> \rightarrow_o <C_{U2}, G, \mathcal{V}> \quad \text{implies} \quad <C_{U1}, C_B \wedge C, \mathcal{V}> \rightarrow_o <C_{U2}, G, \mathcal{V}>.$$

Lemma 2.3 Every built-in constraint solver (where update fulfills the stated requirements) is monotonic.

2.3 Relation between the declarative and the operational semantics

We present results relating the operational and declarative semantics of CHR. These results are based on work of Jaffar and Lassez [JL87], Maher [Mah87] and van Hentenryck [vH91].

Lemma 2.4 Let P be a CHR program, G be a goal. If C is a computable constraint of G, then

$$\mathcal{P}, CT \models \forall\ (C \leftrightarrow G).^5$$

Proof. By induction over the number of computation steps.

Theorem 2.1 (Soundness of successful computations) Let P be a CHR program and G be a goal. If G has a successful computation with final constraint C then

$$\mathcal{P}, CT \models \forall\ (C \leftrightarrow G).$$

Proof. Immediately from lemma 2.4.

The following theorem is stronger than the completeness result presented in [Mah87], in the way that we can reduce the disjunction in the strong completeness theorem to a single disjunct. This is possible, since the computation steps preserve logical equivalence (lemma 2.4).

Theorem 2.2 (Completeness of successful computations) Let P be a CHR program and G be a goal. If $\mathcal{P}, CT \models \forall\ (C \leftrightarrow G)$ and C is satisfiable, then G has a successful computation with final constraint C' such that

$$\mathcal{P}, CT \models \forall\ (C \leftrightarrow C').$$

The next theorem gives a soundness and completeness result for correct CHR programs.

Theorem 2.3 (Soundness and Completeness of failed computations) Let P be a correct CHR program and G be a Goal. The following are equivalent:

a) $\mathcal{P}, CT \models \neg\exists G$
b) G has a finitely failed computation.

3 Confluence of CHR programs

We extend the notion of determinacy as used by Maher in [Mah87] and Saraswat in [Sar93] to CHR by introducing the notion of confluence. The notion of deterministic programs is less expressive and too strict for the CHR formalism,

[5] $\forall F$ is the universal closure of a formula F.

because it is not always possible to transform a CHR program into a deterministic one. This has two reasons, of which the first also holds for the CC formalism:

The constraint system must be closed under negation so that a single-headed CHR program can be transformed into one with non-overlapping guards.

Example 3.1 We want to extend the built-in solver, which contains the built-in constraints $\leq$ and $\doteq$, with a user-defined constraint `maximum(X,Y,Z)` which holds if `Z` is the maximum of `X` and `Y`. The following could be part of a definition for the constraint `maximum`:

```
maximum(X,Y,Z) ⇔ X≤Y | Z≐Y.
maximum(X1,Y1,Z1) ⇔ Y1≤X1 | Z1≐X1.
```

This program cannot be transformed into an equivalent one without overlapping guards.

The second reason is that CHR rules have multiple heads. We can get into a situation, where two rules can be applied to different but overlapping conjunctions of constraints. In general it is not possible to avoid commitment of one of the rules (and thus making the program deterministic[6]) by adding constraints to the guards.

Example 3.2 Consider the following part of a CHR program defining interactions between the boolean operations `not`, `imp` and `or`.

```
not(X,Y), imp(X,Y) ⇔ true | X≐0, Y≐1.
not(X1,Y1), or(X1,Z1,Y1) ⇔ true | X1≐0, Y1≐1, Z1≐1.
```

Note that both rules can be applied to the goal `not(A,B)`$\wedge$`imp(A,B)`$\wedge$`or(A,C,B)`. When we want that only the fist rule can be applied, we have to add a constraint to the guard of the first rule, that `or(A,C,B)` doesn't exist. Such a condition is meta–logical and syntactically not allowed.

In the following we will adopt and extend the terminology and techniques of conditional term rewriting systems (CTRS) [DOS88]. A straightforward translation of results in the field of CTRS was not possible, because the CHR formalism gives rise to phenomena not appearing in CTRS. These include the existence of global knowledge (the built–in constraint store) and local variables.

Definition 3.1 A CHR program is called *terminating*, if there are no infinite computation sequences.

Definition 3.2 Two states S_1 and S_2 are called *joinable* if there exist states S_1', S_2' such that $S_1 \mapsto^* S_1'$ and $S_2 \mapsto^* S_2'$ and S_1' is a variant of S_2' ($S_1' \sim S_2'$).

[6] We extend the notion of deterministic programs to our formalism in the natural way that only one rule can commit by any given goal.

Definition 3.3 A CHR program is called *confluent* if the following holds for all states S, S_1, S_2:

$$\text{If } S \mapsto^* S_1, S \mapsto^* S_2 \text{ then } S_1 \text{ and } S_2 \text{ are joinable.}$$

Definition 3.4 A CHR program is called *locally confluent* if the following holds for all states S, S_1, S_2:

$$\text{If } S \mapsto S_1, S \mapsto S_2 \text{ then } S_1 \text{ and } S_2 \text{ are joinable.}$$

For the following reasoning we require, that rules of a CHR program contain disjoint sets of variables. This requirement means no loss of generality, because every CHR program can be easily transformed into one with disjoint sets of variables.

In order to give a characterization for local confluence we have to introduce the notion of critical pairs:

Definition 3.5 If one or more atoms $H_{i_1}, \ldots, H_{i_k}$ of the head of a CHR rule $H_1, \ldots, H_n \Leftrightarrow G \mid B$ unify with one or more atoms atom $H'_{j_1}, \ldots, H'_{j_k}$ of the head of another or the same CHR rule $H'_1, \ldots, H'_m \Leftrightarrow G' \mid B'$ then the triple

$$(G \wedge G' \wedge H_{i_1} \doteq H'_{j_1} \wedge \ldots \wedge H_{i_k} \doteq H'_{j_k} \mid B \wedge H'_{j_{k+1}} \wedge \ldots \wedge H'_{j_m} =\downarrow= B' \wedge H_{i_{k+1}} \wedge \ldots \wedge H_{i_n} \mid \mathcal{V})$$

is called a *critical pair* of the two CHR rules. $\{i_1, \ldots, i_n\}$ and $\{j_1, \ldots, j_m\}$ are permutations of $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$ respectively, $\mathcal{V}$ is the set of variables appearing in $H_1, \ldots, H_n, H'_1, \ldots, H'_m$.

Example 3.3 Consider example 3.1. There are two trivial[7] and the following nontrivial critical pair:

```
(X≤Y ∧ Y1≤X1 ∧ X≐X1 ∧ Y≐Y1 ∧ Z≐Z1 |
 Z≐Y =↓= Z1≐X1 | [X,Y,Z,X1,Y1,Z1])
```

The rules of example 3.2 have the nontrivial critical pair (We omit the global variable store for reasons of clarity):

```
(X≐X1 ∧ Y≐Y1 |
 imp(X,Y) ∧ X1≐0 ∧ Y1≐1 ∧ Z1≐1 =↓= or(X1,Z1,Y1) ∧ X≐0 ∧ Y≐1 | [..])
```

Trivial critical pairs in example 3.1 are stemming from unifying the heads of either the first or second rule with themselves. Note that not every critical pair stemming from one rule only is trivial. If the head of a rule contains a constraint symbol more than once, the resulting critical pair may be nontrivial.

Definition 3.6 A critical pair $(G \mid B_1 =\downarrow= B_2 \mid \mathcal{V})$ is called *joinable* if $<B_1, G, \mathcal{V}>$ and $<B_2, G, \mathcal{V}>$ are joinable.

[7] We call critical pairs of the form $(G \mid B =\downarrow= B \mid \mathcal{V})$ trivial.

Example 3.4 The first critical pair in example 3.3 is joinable, if the built–in constraint solver simplifies $\mathtt{X}\leq\mathtt{Y} \wedge \mathtt{Y}\leq\mathtt{X}$ to the constraint $\mathtt{X}\doteq\mathtt{Y}$.

The following lemmas are necessary to prove theorem 3.2. The proofs for these lemmas can be found in the appendix. The first lemma states that the global variables are not touched when testing the variance of two states. Crucial for this lemma is the fact that $\mathcal{V}$ is an ordered set.

Lemma 3.1 If

$$<C_{U1}, C_{B1}, \mathcal{V}> \sim <C_{U2}, C_{B2}, \mathcal{V}>$$

then the variables in $\mathcal{V}$ are not modified by variable renaming.

The following lemma shows that enclosement guarantees that addition of built–in constraints is compatible with update:

Lemma 3.2 If $(C, C_U \wedge C_B)$ is enclosed by $\mathcal{V}$ and

$$\begin{array}{rcl} <C_U, C_B, \mathcal{V}> & \mapsto^* & <C'_U, C'_B, \mathcal{V}> \quad \text{then} \\ <C_U, C_B \wedge C, \mathcal{V}> & \mapsto^* & \text{update}(<C'_U, C'_B \wedge C, \mathcal{V}>). \end{array}$$

We apply lemma 3.2 to prove lemma 3.3, stating the enclosement conditions under which joinability of states is compatible with addition of built–in constraints.

Lemma 3.3 If

$$\begin{array}{rcl} <C_{U1}, C_{B1}, \mathcal{V}> & \mapsto^* & <C'_{U1}, C'_{B1}, \mathcal{V}>, \\ <C_{U2}, C_{B2}, \mathcal{V}> & \mapsto^* & <C'_{U2}, C'_{B2}, \mathcal{V}>, \\ <C'_{U1}, C'_{B1}, \mathcal{V}> & \sim & <C'_{U2}, C'_{B2}, \mathcal{V}>, \end{array}$$

and $(C, C_{U1} \wedge C_{B1})$ and $(C, C_{U2} \wedge C_{B2})$ are enclosed by $\mathcal{V}$, then

a)

$$\begin{array}{rcl} <C_{U1}, C_{B1} \wedge C, \mathcal{V}> & \mapsto^* & \text{update}(<C'_{U1}, C'_{B1} \wedge C, \mathcal{V}>), \\ <C_{U2}, C_{B2} \wedge C, \mathcal{V}> & \mapsto^* & \text{update}(<C'_{U2}, C'_{B2} \wedge C, \mathcal{V}>), \\ \text{update}(<C'_{U1}, C'_{B1} \wedge C, \mathcal{V}>) & \sim & \text{update}(<C'_{U2}, C'_{B2} \wedge C, \mathcal{V}>), \end{array}$$

b)

$$\begin{array}{rcl} <C_{U1} \wedge C, C_{B1}, \mathcal{V}> & \mapsto^* & \text{update}(<C'_{U1} \wedge C, C'_{B1}, \mathcal{V}>), \\ <C_{U2} \wedge C, C_{B2}, \mathcal{V}> & \mapsto^* & \text{update}(<C'_{U2} \wedge C, C'_{B2}, \mathcal{V}>), \\ \text{update}(<C'_{U1} \wedge C, C'_{B1}, \mathcal{V}>) & \sim & \text{update}(<C'_{U2} \wedge C, C'_{B2}, \mathcal{V}>). \end{array}$$

Definition 3.7 We call two states $<C_{U1}, C_{B1}, \mathcal{V}>$ and $<C_{U2}, C_{B2}, \mathcal{V}>$ *update equivalent* iff

$$\mathrm{update}(<C_{U1}, C_{B1}, \mathcal{V}>) = \mathrm{update}(<C_{U2}, C_{B2}, \mathcal{V}>)$$

Lemma 3.4 If $<C_U, C_B, \mathcal{V}>$ and $<C'_U, C'_B, \mathcal{V}>$ are update equivalent and $<C_U, C_B, \mathcal{V}> \mapsto^* S'$, then $<C'_U, C'_B, \mathcal{V}> \mapsto^* S'$.

Proof. The lemma follows directly from lemma 2.2.

The next lemma gives a condition when joinability is compatible with changing the global variable store:

Lemma 3.5 Let $<C_{U1}, C_{B1}, \mathcal{V}>$ and $<C_{U2}, C_{B2}, \mathcal{V}>$ be joinable. Then the following holds:

a) $<C_{U1}, C_{B1}, \mathcal{V}'>$ and $<C_{U2}, C_{B2}, \mathcal{V}'>$ are joinable,
   if $\mathcal{V}'$ consists only of variables contained in $\mathcal{V}$.

b) $<C_{U1}, C_{B1}, \mathcal{V} \circ \mathcal{V}'>$ and $<C_{U2}, C_{B2}, \mathcal{V} \circ \mathcal{V}'>$ are joinable,
   if $\mathcal{V}'$ contains only fresh variables ($\circ$ denotes concatenation).

The following theorem is an analogy to Newman's Lemma for term rewriting systems [Pla93] and is proven analogously:

Theorem 3.1 (confluence of CHR programs) If a CHR program is locally confluent and terminating, it is confluent.

Theorem 3.2 gives a characterization for locally confluent CHR programs. The proof is given in the appendix and relies on lemmas 3.1 to 3.5.

Theorem 3.2 (local confluence of CHR programs) A terminating CHR program is locally confluent if and only if all its critical pairs are joinable.

The theorem also means that we can decide whether a program (which we do not know is terminating or not) will be confluent in case it is terminating.

Example 3.5 This example illustrates the case that an unjoinable critical pair is detected. The following CHR program is an implementation of `merge/3`, i.e. merging two lists into one list as the elements of the input lists arrive. Thus the order of elements in the final list can differ from computation to computation.

```
merge([],L2,L3) ⇔ true | L2≐L3.
merge(M1,[],M3) ⇔ true | M1≐M3.
merge([X|N1],N2,N3) ⇔ true | N3≐[X|N], merge(N1,N2,N).
merge(O1,[Y|O2],O3) ⇔ true | O3≐[Y|O], merge(O1,O2,O).
```

There are 8 critical pairs, 4 of them stemming from different rules.
If `merge/3` meets the specification, there is space for nondeterminism that causes non-confluence. Indeed, a look at the critical pairs reveals one critical pair stemming from the third and fourth rule that is not joinable:

```
([X|N1]≐O1 ∧ N2≐[Y|O2] ∧ N3≐O3 |
 N3≐[X|N] ∧ merge(N1,N2,N) =↓= O3≐[Y|O] ∧ merge(O1,O2,O) | [..])
```

It can be seen from the unjoinable critical pair above that a state like `<merge([a],[b],L),true,[L]>` can either result in putting `a` before `b` in the output list `L` or vice versa, since a **Simplify**–step can result in differing unjoinable states, depending on which rule is applied. Hence - not surprisingly - `merge/3` is not confluent.

4 Correctness and Confluence of CHR Programs

Definition 4.1 Given a CHR program P, we define the computation equivalence $\leftrightarrow_P^*$: $S_1 \leftrightarrow_P S_2$ iff $S_1 \mapsto S_2$ or $S_1 \mapsfrom S_2$. $S \leftrightarrow_P^* S'$ iff there is a sequence $S_1, \ldots S_n$ such that S_1 is S, S_n is S' and $S_i \leftrightarrow_P S_{i+1}$ for all i. We will write $\leftrightarrow$ instead of $\leftrightarrow_P$ and $\leftrightarrow^*$ instead of $\leftrightarrow_P^*$, if the program P is fixed.

For the sake of simplicity and clarity we prove the following two lemmas only for the special case that all rules are ground–instantiated, without guards and that *true* and *false* are the only built–in constraints used. One can extend the proof to full CHR by transforming each rule of a CHR program into (possibly infinitely many) ground–instantiated rules. This includes evaluating the built–in constraints in the guards and bodies.

Lemma 4.1 If P is confluent, then $<\mathrm{true},\mathrm{true},\mathcal{V}> \leftrightarrow_P^* <\mathrm{true},\mathrm{false},\mathcal{V}>$ does not hold.

Proof. We show by induction on n that there are no states $S_1, T_1, S_2, \ldots, T_{n-1}, S_n$ such that

$$<\mathrm{true},\mathrm{true},\mathcal{V}> {}^*\!\!\mapsfrom S_1 \mapsto^* T_1 {}^*\!\!\mapsfrom S_2 \mapsto^* \ldots \mapsto^* T_{n-1} {}^*\!\!\mapsfrom S_n \mapsto^* <\mathrm{true},\mathrm{false},\mathcal{V}>$$

Base case: $<\mathrm{true},\mathrm{true},\mathcal{V}> {}^*\!\!\mapsfrom S_1 \mapsto^* <\mathrm{true},\mathrm{false},\mathcal{V}>$ cannot exist, because $<\mathrm{true},\mathrm{true},\mathcal{V}>$ and $<\mathrm{true},\mathrm{false},\mathcal{V}>$ are different (no variants) final states and P is confluent.

Induction step: We assume that the induction hypothesis holds for n, i.e. $<\mathrm{true},\mathrm{true},\mathcal{V}> {}^*\!\!\mapsfrom S_1 \mapsto^* T_1 {}^*\!\!\mapsfrom S_2 \mapsto^* \ldots \mapsto^* T_{n-1} {}^*\!\!\mapsfrom S_n \mapsto^* <\mathrm{true},\mathrm{false},\mathcal{V}>$ doesn't exist. We prove the assertion for $n+1$ by contradiction:
We assume that a sequence of the form $<\mathrm{true},\mathrm{true},\mathcal{V}> {}^*\!\!\mapsfrom S_1 \mapsto^* T_1 {}^*\!\!\mapsfrom S_2 \mapsto^* T_2 {}^*\!\!\mapsfrom \ldots {}^*\!\!\mapsfrom S_n \mapsto^* T_n {}^*\!\!\mapsfrom S_{n+1} \mapsto^* <\mathrm{true},\mathrm{false},\mathcal{V}>$ exists. We will lead this assumption to a contradiction.

Since P is confluent, $<\mathrm{true},\mathrm{false},\mathcal{V}>$ and T_n are joinable. Since $<\mathrm{true},\mathrm{false},\mathcal{V}>$ is a final state, there is a computation of T_n that results in $<\mathrm{true},\mathrm{false},\mathcal{V}>$ ($T_n \mapsto^* <\mathrm{true},\mathrm{false},\mathcal{V}>$), and hence $S_n \mapsto^* <\mathrm{true},\mathrm{false},\mathcal{V}>$. Therefore there is a sequence of the form

$<\mathrm{true},\mathrm{true},\mathcal{V}> {}^*\!\!\mapsfrom S_1 \mapsto^* T_1 {}^*\!\!\mapsfrom S_2 \mapsto^* T_2 {}^*\!\!\mapsfrom \ldots {}^*\!\!\mapsfrom S_{n-1} \mapsto^* T_{n-1} {}^*\!\!\mapsfrom S_n \mapsto^*$ $<\mathrm{true},\mathrm{false},\mathcal{V}>$,

which is a contradiction to the induction hypothesis.

Lemma 4.2 If $<\mathrm{true}, \mathrm{true}, \mathcal{V}> \leftrightarrow^* <\mathrm{true}, \mathrm{false}, \mathcal{V}>$ does not hold, then $\mathcal{P} \cup CT$ is consistent.

Proof. We show consistency by defining an interpretation which is a model of $\mathcal{P}$, and therefore of $\mathcal{P} \cup CT$.

We define $I_0 := \{\{C_1, \ldots, C_n\} | <C_1 \wedge \ldots \wedge C_n, \mathrm{true}, \mathcal{V}> \leftrightarrow^* <\mathrm{true}, \mathrm{true}, \mathcal{V}>\}$. Let be $I := (\bigcup I_0) \backslash \{\mathrm{true}\}$ ($\bigcup M$ is the union of all members of M). false $\notin I$, because $<\mathrm{false}, \mathrm{true}, \mathcal{V}> \leftrightarrow^* <\mathrm{true}, \mathrm{true}, \mathcal{V}>$ does not hold. Therefore I is a Herbrand interpretation.

We show that $I \models \mathcal{P}$:
For all formulas $H_1 \wedge \ldots \wedge H_n \leftrightarrow B_1 \wedge \ldots \wedge B_m \in \mathcal{P}$ the following equivalences hold:

$$
\begin{array}{ll}
 & I \models H_1 \wedge \ldots \wedge H_n \\
\text{iff} & \{H_1, \ldots, H_n\} \subseteq I \\
\text{iff} & <H_1 \wedge \ldots \wedge H_n, \mathrm{true}, \mathcal{V}> \leftrightarrow^* <\mathrm{true}, \mathrm{true}, \mathcal{V}> \\
\text{iff} & <B_1 \wedge \ldots \wedge B_m, \mathrm{true}, \mathcal{V}> \leftrightarrow^* <\mathrm{true}, \mathrm{true}, \mathcal{V}> \\
\text{iff} & \{B_1, \ldots, B_m\} \subseteq I \\
\text{iff} & I \models B_1 \wedge \ldots \wedge B_m.
\end{array}
$$

Therefore $I \models H_1 \wedge \ldots \wedge H_n \leftrightarrow B_1 \wedge \ldots \wedge B_m$ for all formulas $H_1 \wedge \ldots \wedge H_n \leftrightarrow B_1 \wedge \ldots \wedge B_m$ in $\mathcal{P}$.

Theorem 4.1 If P is confluent, then $\mathcal{P} \cup CT$ is consistent.

Proof. The theorem follows directly from the lemmas 4.1 and 4.2.

Maher proves the following result for deterministic programs: if any computation sequence terminates in failure, then every (fair) computation sequence terminates in failure. We extend this result on confluent programs and give, compared to theorem 2.3, a closer relation between the operational and declarative semantics.

Definition 4.2 A computation is *fair* iff the following holds:
If a rule can be applied infinitely often to a goal, then it is applied at least once.

Lemma 4.3 Let P be a confluent CHR program and G be a goal which has a finitely failed derivation. Then every fair derivation of G is finitely failed.

The following theorem is a consequence of the above lemma and theorem 2.3.

Theorem 4.2 Let P be a confluent program and G be a Goal.
The following are equivalent:

a) $\mathcal{P}, CT \models \neg \exists G$
b) G has a finitely failed computation.
c) every fair computation of G is finitely failed.

5 Conclusion and Future Work

We introduced the notion of confluence for Constraint Handling Rules (CHR). Confluence guarantees that a CHR program will always compute the same result for a given set of user-defined constraints independent of which rules are applied.

We have given a characterization of confluent CHR programs through joinability of critical pairs, yielding a decidable, syntactically based test for confluence. We have shown that confluence is a sufficient condition for logical correctness of CHR programs. Correctness is an essential property of constraint solvers.

We also gave various soundness and completeness results for CHR programs. Some of these theorems are stronger than what holds for the related families of (concurrent) constraint programming languages due to correctness.

Our approach complements recent work [MO95] that gives confluent, non-standard semantics for CC languages to make them amenable to abstract interpretation and analysis in general, since our confluence test can find out parts of CC programs which are confluent already under the standard semantics.

Current work integrates the two other kinds of CHR rules, the propagation and the simpagation rules, into our condition for confluence. We are also developing a tool in ECL^{i}PS^e(ECRC Constraint Logic Programming System [Ecl94]) which tests confluence of CHR programs. Preliminary tests show that most existing constraint solvers written in CHR are indeed confluent, but that there are inherently non-confluent solvers (e.g. performing Gaussian elimination), too. We also plan to investigate completion methods to make a non-confluent CHR program confluent.

Acknowledgements

We would like to thank Heribert Schütz and Norbert Eisinger for useful comments on a preliminary version of this paper and Michael Marte for his implementation of a confluence tester.

References

[BFL+94] P. Brisset, T. Frühwirth, P. Lim, M. Meier, T. Le Provost, J. Schimpf, and M. Wallace. *ECL^{i}PS^e 3.4 Extensions User Manual.* ECRC Munich Germany, July 1994.

[Ecl94] *ECL^{i}PS^e 3.4 User Manual*, July 1994.

[FGMP95] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence in concurrent constraint programming. In Alagar and Nivat, editors, *Proceedings of AMAST '95, LNCS 936.* Springer, 1995.

[Frü95] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends.* LNCS 910, March 1995.

[JL87] J. Jaffar and J. L. Lassez. Constraint logic programming. In *Proceedings of the 14^{th} ACM Symposium on Principles of Programming Languages POPL-87, Munich, Germany*, pages 111–119, 1987.

[JM94] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 20:503–581, 1994.

[KK91] Claude Kirchner and Hélène Kirchner. *Rewriting: Theory and Applications.* North-Holland, 1991.

[Mah87] M. J. Maher. Logic Semantics for a Class of Committed-Choice Programs. In *Fourth International Conference on Logic Programming*, pages 858–876, Melbourne, Australia, May 1987.

[MO95] K. Marriott and M. Odersky. A confluent calculus for concurrent constraint programming with guarded choice. In Ugo Montanari Francesca Rossi, editor, *Principles and Practice of Constraint Programming, Proceedings First International Conference, CP'95, Cassis, France*, pages 310–327, Berlin, September 1995. Springer.

[Pla93] David A. Plaisted. Equational reasoning and term rewriting systems. In D. Gabbay, C. Hogger, J. A. Robinson, and J. Siekmann, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1, chapter 5, pages 273–364. Oxford University Press, Oxford, 1993.

[Sar93] V. A. Saraswat. *Concurrent Constraint Programming.* MIT Press, Cambridge, 1993.

[Sha89] E. Shapiro. The family of concurrent logic programming languages. In *ACM Computing Surveys*, volume 21:3, pages 413–510, September 1989.

[SRP91] V. A. Saraswat, M. Rinard, and P. Panangaden. The semantics foundations of concurrent constraint programming. In *Conference Record of the Eighteenth Annual ACM Symposium on principles of Programming Languages*, pages 333–352, Orlando, Florida, January 1991. ACM Press.

[vH91] P. van Hentenryck. Constraint logic programming. In *The Knowledge Engineering Review*, volume 6, pages 151–194, 1991.

[DOS88] N. Dershowitz, N. Okada, and G. Sivakumar. Confluence of conditional rewrite systems. In *1st CTRS*, pages 31–44. LNCS 308, 1988.

Appendix: Proofs for section 3

This appendix may be omitted in the final version in case of space limitations.

In this appendix we prove lemmas 3.1, 3.2, 3.3, 3.5 and theorem 3.2.

Proof of lemma 3.1. Let $<C_{U1}, C_{B1}, \mathcal{V}>\sigma = <C_{U2}, C_{B2}, \mathcal{V}>$ and $<C_{U1}, C_{B1}, \mathcal{V}> = <C_{U2}, C_{B2}, \mathcal{V}>\tau$, where σ and τ are substitutions. Then $\mathcal{V}\sigma = \mathcal{V}$ and $\mathcal{V} = \mathcal{V}\tau$ and therefore for all variables X in the ordered set $\mathcal{V}$, $X\sigma = X\tau = X$ holds.

Proof of lemma 3.2. Let $S = <C_U, C_B, \mathcal{V}>$, $S' = <C'_U, C'_B, \mathcal{V}>$. We prove the claim by induction over the number n of computation steps applied:
$n = 0$**:** Then update$(S) = S'$ or $S = S'$ and therefore update$(<C_U, C_B \wedge C, \mathcal{V}>) =$ update$(<C'_U, C'_B \wedge C, \mathcal{V}>)$, because of lemma 2.1 and (C, C_B) being enclosed in $\mathcal{V}$.
$n+1$**:** Let $S \mapsto <C_{U1}, C_{B1}, \mathcal{V}> \mapsto^* S'$. Then the first computation step is also applicable to $<C_U, C_B \wedge C, \mathcal{V}>$, because the constraint solver is monotonic (and therefore all rules applicable to S are also applicable to $<C_U, C_B \wedge C, \mathcal{V}>$). Assume that $<C_U, C_B \wedge C, \mathcal{V}> \not\mapsto$ update$(<C_{U1}, C_{B1} \wedge C, \mathcal{V}>)$. Then there are constraints in C_B interfering with C, that have been deleted by update due to locality of equality constraints. If the constraints interfere, they must share variables which we know to be contained in $\mathcal{V}$. Therefore the equality constraints could not be local and we have a contradiction.
The induction hypothesis gives that $<C_{U1}, C_{B1} \wedge C, \mathcal{V}> \mapsto^*$ update$(<C'_U, C'_B \wedge C, \mathcal{V}>)$. Because update has no influence on application of rules (lemma 2.2) we conclude:

$$<C_U, C_B \wedge C, \mathcal{V}> \mapsto \text{update}(<C_{U1}, C_{B1} \wedge C, \mathcal{V}>) \mapsto^* \text{update}(<C'_U, C'_B \wedge C, \mathcal{V}>).$$

Proof of lemma 3.3. a) Because of lemma 3.2 we only have to show that update$(<C'_{U1}, C'_{B1} \wedge C, \mathcal{V}>) \sim$ update$(<C'_{U2}, C'_{B2} \wedge C, \mathcal{V}>)$: Let

$$<C'_{U1}, C'_{B1}, \mathcal{V}>\sigma = <C'_{U2}, C'_{B2}, \mathcal{V}> \text{ and } <C'_{U1}, C'_{B1}, \mathcal{V}> = <C'_{U2}, C'_{B2}, \mathcal{V}>\tau.$$

σ and τ don't touch variables in C (lemma 3.1) therefore we have $C\sigma = C$ and $C\tau = C$. Hence we can conclude that $<C'_{U1}, C'_{B1} \wedge C, \mathcal{V}>\sigma = <C'_{U2}, C'_{B2} \wedge C, \mathcal{V}>$ and $<C'_{U1}, C'_{B1} \wedge C, \mathcal{V}> = <C'_{U2}, C'_{B2} \wedge C, \mathcal{V}>\tau$, i.e. $<C'_{U1}, C'_{B1} \wedge C, \mathcal{V}>$ and $<C'_{U2}, C'_{B2} \wedge C, \mathcal{V}>$ are variants, and therefore

$$\text{update}(<C'_{U1}, C'_{B1} \wedge C, \mathcal{V}>) \sim \text{update}(<C'_{U2}, C'_{B2} \wedge C, \mathcal{V}>).$$

b) It is easy to see that $<C_{U1} \wedge C, C_{B1}, \mathcal{V}> \mapsto^*$ update$(<C'_{U1} \wedge C, C'_{B1}, \mathcal{V}>)$ and $<C_{U2} \wedge C, C_{B2}, \mathcal{V}> \mapsto^*$ update$(<C'_{U2} \wedge C, C'_{B2}, \mathcal{V}>)$ holds. The same argumentation as above holds here to prove the variance of update$(<C'_{U1} \wedge C, C'_{B1}, \mathcal{V}>)$ and update$(<C'_{U2} \wedge C, C'_{B2}, \mathcal{V}>)$.

Proof of lemma 3.5. Let

$$\begin{aligned} <C_{U1}, C_{B1}, \mathcal{V}> &\mapsto^* <C'_{U1}, C'_{B1}, \mathcal{V}>, \\ <C_{U2}, C_{B2}, \mathcal{V}> &\mapsto^* <C'_{U2}, C'_{B2}, \mathcal{V}>, \\ <C'_{U1}, C'_{B1}, \mathcal{V}> &\sim <C'_{U2}, C'_{B2}, \mathcal{V}>. \end{aligned}$$

a) We know that the following holds:

$$<C_{U1}, C_{B1}, \mathcal{V}'> \mapsto^* <C'_{U1}, C''_{B1}, \mathcal{V}'>, <C_{U2}, C_{B2}, \mathcal{V}'> \mapsto^* <C'_{U2}, C''_{B2}, \mathcal{V}'>$$

where C''_{B1} is C'_{B1} with local equality constraints removed by update (due to shrinking $\mathcal{V}$ to $\mathcal{V}'$ variables may change their status from global to local). The

same holds for C''_{B2} and C'_{B2}. Therefore C''_{B1} and C''_{B2} must be variants, because C'_{B1} and C'_{B2} were variants, and C''_{B1} and C''_{B2} derive from C'_{B1} and C'_{B2} by deletion of the same local equality constraints.
b) We can apply the same computation steps to $<C_{U1}, C_{B1}, \mathcal{V} \circ \mathcal{V}'>$ as to $<C_{U1}, C_{B1}, \mathcal{V}>$ and therefore have $<C_{U1}, C_{B1}, \mathcal{V}> \mapsto^* <C'_{U1}, C'_{B1}, \mathcal{V} \circ \mathcal{V}'>$. With exactly the same argumentation we have that $<C_{U2}, C_{B2}, \mathcal{V}> \mapsto^* <C'_{U2}, C'_{B2}, \mathcal{V} \circ \mathcal{V}'>$. $<C'_{U1}, C'_{B1}, \mathcal{V} \circ \mathcal{V}'>$ and $<C'_{U2}, C'_{B2}, \mathcal{V} \circ \mathcal{V}'>$ are variants.

Proof of theorem 3.2. For the only if direction we assume that we have a locally confluent CHR program with a critical pair, that is not joinable. We will lead this assumption to a contradiction:
Let

$$(G \mid C_1 \ \ {=}{\downarrow}{=} \ \ C_2 \mid \mathcal{V})$$

be a non joinable critical pair, i.e. the computations of $<G, C_1, \mathcal{V}>$ and $<G, C_2, \mathcal{V}>$ result in two final states, not being variants. Because $<G, C_1, \mathcal{V}>$ and $<G, C_2, \mathcal{V}>$ derive from a common state S by one application of **Solve** and **Simplify**, this contradicts the local confluence of the program.

In order to prove the if direction we prove the following hypothesis by structural induction over the states. The wellfounded ordering used for induction is induced by the (terminating) application of **Solve** and **Simplify**.
Induction hypothesis:

$$\begin{array}{c} \text{for all states } <C_U, C_B, \mathcal{V}> \\ \text{the following implication holds for all } C_{U1}, C_{U2}, C_{B1}, C_{B2}: \\ <C_U, C_B, \mathcal{V}> \mapsto <C_{U1}, C_{B1}, \mathcal{V}> \text{ and } <C_U, C_B, \mathcal{V}> \mapsto <C_{U2}, C_{B2}, \mathcal{V}> \\ \text{implies} \quad <C_{U1}, C_{B1}, \mathcal{V}> \text{ and } <C_{U2}, C_{B2}, \mathcal{V}> \text{ are joinable.} \end{array}$$

The base case of the induction is trivial: we are in a situation where there are no rules applicable to $<C_U, C_B, \mathcal{V}>$. The antecedent of our induction hypothesis turns out to be false in this situation, therefore the implication is true.

Now assume that we are in state $<C_U, C_B, \mathcal{V}>$ where there are two or more possibilities of computation. We investigate all pairs of two possibilities and show that they are joinable.

There are three cases:
Case. Solve+Solve: We required the constraint theory CT to preserve commutativity of the conjunction $\wedge$ in the built–in store, therefore the two computations will result in identical states.
Case. Solve+Simplify: We are in a situation $<C \wedge H' \wedge C_U, C_B, \mathcal{V}>$ where C is a built–in constraint, and H' is a conjunction of user–defined constraints matching with the head of a rule $(H \Leftrightarrow G \mid B)$ and the guard is entailed, i.e.

$$<H', C_B, \mathcal{V}> \rightarrow_o <H', H' \doteq H \wedge G, \mathcal{V}>.$$

If the **Simplify** step is done first, the **Solve** step is applicable hereafter. If we show, that **Simplify** is applicable after application of **Solve**, we are finished. Application of **Solve** results in a state update($<C \wedge H' \wedge C_U, C \wedge C_B, \mathcal{V}>$). **Simplify** is applicable if $<H', C \wedge C_B, \mathcal{V}> \rightarrow_o <H', H' \doteq H \wedge G, \mathcal{V}>$. This holds because the built–in constraint solver is monotonic.

Case. Simplify+Simplify: Rules to be applied (we assume that they contain disjoint sets of variables):

$$\begin{array}{rcl} R & \equiv & H_1, \ldots, H_n \;\Leftrightarrow\; G \mid B \\ R' & \equiv & H'_1, \ldots, H'_{n'} \;\Leftrightarrow\; G' \mid B' \end{array}$$

It is explicitly allowed, that R and R' are in fact variants of the same CHR rule.

We have to show that application of R or R' results in joinable states. We know that C_B is satisfiable, otherwise no rule could be applied. We can distinguish two different subcases:

Subcase. Disjoint peak: No head atom of the first rule R unifies with a head atom of the other rule R'.
Application order $R; R'$ (first R then R') will lead to the same result as application order $R'; R$. Because of the associativity and commutativity of the goal store both orderings are indeed applicable.
Subcase. Critical peak. We can assume that the first atoms of the rules unify ($H_1 \doteq H'_1 \wedge \ldots \wedge H_i \doteq H'_i$). (Included in this case is the overlay case: $n = n' \wedge H_1 \doteq H'_1 \wedge \ldots \wedge H_n \doteq H'_n$.)
Let $<G_1 \wedge G_2 \wedge \ldots \wedge G_m, C_B, \mathcal{V}>$ be the actual state, on which the rules R and R' are applicable. For the sake of simplicity we can assume that

$$<G_1 \wedge G_2 \wedge \ldots \wedge G_m, C_B, \mathcal{V}> \rightarrow_o <G_1 \wedge G_2 \wedge \ldots \wedge G_m, C, \mathcal{V}>,$$

where C is the following conjunction of equality constraints:

$$\begin{array}{rl} C \equiv & G_1 \doteq H_1 \;\wedge \ldots \wedge\; G_i \doteq H_i \;\wedge\; G_1 \doteq H'_1 \;\wedge \ldots \wedge\; G_i \doteq H'_i \;\wedge \\ & G_{i+1} \doteq H_{i+1} \;\wedge \ldots \wedge\; G_n \doteq H_n \;\wedge\; G_{n+1} \doteq H'_{i+1} \wedge \ldots \wedge\; G_{n+n'-i} \doteq H'_{n'}. \end{array}$$

(This situation can be achieved by changing the order of goal atoms and atoms in the heads of R and R'.)
We use abbreviations to represent the atoms in question:

$$\begin{array}{rcl} \bar{G} & = & G_1 \wedge \ldots \wedge G_n, \\ \bar{G}' & = & G_1 \wedge \ldots \wedge G_i \wedge G_{n+1} \wedge \ldots \wedge G_{n+n'-i}, \\ \bar{H} & = & H_1 \wedge \ldots \wedge H_n, \\ \bar{H}' & = & H'_1 \wedge \ldots \wedge H'_{n'}, \\ H_\cap & = & H_1 \wedge \ldots \wedge H_i, \\ H'_\cap & = & H'_1 \wedge \ldots \wedge H'_i, \\ G_{REST} & = & G_{n+1} \wedge \ldots \wedge G_{n+n'-i+1} \wedge \ldots \wedge G_m, \\ G'_{REST} & = & G_{i+1} \wedge \ldots \wedge G_n \wedge G_{n+n'-i+1} \wedge \ldots \wedge G_m. \end{array}$$

and write $(P_1, \ldots, P_p) \doteq (P'_1, \ldots, P'_p)$ for $P_1 \doteq P'_1 \wedge \ldots \wedge P_p \doteq P'_p$.
Application of R and R' on the actual state will result in the following two states:

$$\begin{array}{rrl} & S = & \text{update}(<G_{REST} \wedge B, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{V}>) \\ \text{and} & S' = & \text{update}(<G'_{REST} \wedge B', C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}>). \end{array}$$

We will show in the following that S and S' are joinable:
We can see that the rules R and R' have the critical pair

$$(G \wedge G' \wedge H_\cap \doteq H'_\cap \mid B \wedge H'_{i+1} \wedge \ldots \wedge H'_{n'} =\downarrow= B' \wedge H_{i+1} \wedge \ldots \wedge H_n \mid \mathcal{V}'),$$

which we know to be joinable. So there is a sequence of computation steps resulting in two final states differing only by variable renaming for the two initial states

$$\begin{array}{rl} & <B \wedge H'_{i+1} \wedge \ldots \wedge H'_{n'}, G \wedge G' \wedge H_\cap \doteq H'_\cap, \mathcal{V}'> \\ \text{and} & <B' \wedge H_{i+1} \wedge \ldots \wedge H_n, G \wedge G' \wedge H_\cap \doteq H'_\cap, \mathcal{V}'>. \end{array}$$

We can apply lemma 3.5b) here and add $\mathcal{V}$ to the global variables stores, because $\mathcal{V}$ shares no variables with the two states:

$$\begin{array}{rl} & <B \wedge H'_{i+1} \wedge \ldots \wedge H'_{n'}, G \wedge G' \wedge H_\cap \doteq H'_\cap, \mathcal{V}' \circ \mathcal{V}> \\ \text{and} & <B' \wedge H_{i+1} \wedge \ldots \wedge H_n, G \wedge G' \wedge H_\cap \doteq H'_\cap, \mathcal{V}' \circ \mathcal{V}>. \end{array}$$

are joinable.
According to lemma 3.3 a) and b) we can add the built-in constraints $C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}'$ and the user-defined constraints $G_{n+1} \wedge \ldots \wedge G_m$ to the constraint stores of each state without losing joinability. The requirements of the lemma are met because the variables in $\bar{H}$ and $\bar{H}'$ are contained in $\mathcal{V}'$ and $C_B, \bar{G}, \bar{G}'$ and $G_{n+1} \wedge \ldots \wedge G_m$ share no variables with the previous states.

$$\begin{array}{rl} & <BH'G, G \wedge G' \wedge H_\cap \doteq H'_\cap \wedge C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}' \circ \mathcal{V}> \\ \text{and} & <B'HG, G \wedge G' \wedge H_\cap \doteq H'_\cap \wedge C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}' \circ \mathcal{V}> \end{array}$$

are joinable. Here $BH'G$ stands for $B \wedge H'_{i+1} \wedge \ldots \wedge H'_{n'} \wedge G_{n+1} \wedge \ldots \wedge G_m$ and $B'HG$ stands for $B' \wedge H_{i+1} \wedge \ldots \wedge H_n \wedge G_{n+1} \wedge \ldots \wedge G_m$.
Now we can remove the global variables $\mathcal{V}'$ from the variable stores by applying lemma 3.5a) and keep joinability of

$$\begin{array}{rl} & <BH'G, G \wedge G' \wedge H_\cap \doteq H'_\cap \wedge C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}> \\ \text{and} & <B'HG, G \wedge G' \wedge H_\cap \doteq H'_\cap \wedge C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}>. \end{array}$$

We know that

$$<G_1 \wedge \ldots \wedge G_m, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}> \rightarrow_o <G_1 \wedge \ldots \wedge G_m, H_\cap \doteq H'_\cap \wedge G \wedge G', \mathcal{V}>,$$

hence

$$\begin{array}{rl} \text{update}(<BH'G, G \wedge G' \wedge H_\cap \doteq H'_\cap \wedge C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}>) & = \\ \text{update}(<BH'G, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}>). & \end{array}$$

With the analogous reasoning for

$$<B'HG, G \wedge G' \wedge H_\cap \doteq H'_\cap \wedge C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}>$$

and applying lemma 3.4 we can remove $G \wedge G' \wedge H_\cap \doteq H'_\cap$ in both states, whilst keeping joinability of

$$\begin{array}{rl} & <BH'G, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}> \\ \text{and} & <B'HG, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}>. \end{array}$$

We can replace $H'_{i+1} \wedge \ldots \wedge H'_{n'}$ by $G_{n+1} \wedge \ldots \wedge G_{n+n'-i}$ and likewise $H_{i+1} \wedge \ldots \wedge H_n$ by $G_{i+1} \wedge \ldots \wedge G_n$ and get (joinable) update-equivalent states, because the equality-constraints are contained in the constraint store:

$$
\begin{array}{ll}
 & <B \wedge G_{REST}, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}> \\
\text{and} & <B' \wedge G'_{REST}, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}>
\end{array}
$$

We know that the following two states are update-equivalent with the upper two states (only local equality constraints have been removed) and therefore must be joinable (lemma 3.4), too.

$$
\begin{array}{ll}
 & <B \wedge G_{REST}, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{V}> \\
\text{and} & <B' \wedge G'_{REST}, C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}>
\end{array}
$$

If we apply lemma 3.4 we finally know that S and S' are joinable.