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

Ludwig
Maximilians–
Universität
München

Operational Semantics and Confluence of Constraint Propagation Rules

Slim Abdennadher

In: Proc. Third International Conference on Principles and Practice of Constraint Programming, Schloss Hagenberg, Austria, Springer LNCS, 1997.
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1997-6, Oktober 1997

Operational Semantics and Confluence of Constraint Propagation Rules

Slim Abdennadher

Computer Science Department, University of Munich
Oettingenstr. 67, 80538 München, Germany
Slim.Abdennadher@informatik.uni-muenchen.de

Abstract. Constraint Handling Rules (CHR) allow one to specify and implement both propagation and simplification for user-defined constraints. Since a propagation rule is applicable again and again, we present in this paper for the first time an operational semantics for CHR that avoids the termination problem with propagation rules.
In previous work [AFM96], a sufficient and necessary condition for the confluence of terminating simplification rules was given inspired by results about conditional term rewriting systems. Confluence ensures that the solver will always compute the same result for a given set of constraints independent of which rules are applied. The confluence of propagation rules was an open problem. This paper shows that we can also give a sufficient and a necessary condition for confluence of terminating CHR programs with propagation rules based on the more refined operational semantics.

1 Introduction

Constraint Logic Programming [vH91, JM94] combines the declarativity of logic programming with the efficiency of constraint solving. As it runs, a constraint-based program successively generates pieces of partial information called constraints. The constraint solver has the task to collect, combine, and simplify them and to detect their inconsistency.

Constraint Handling Rules

Constraint handling rules (CHR) [Frü95] are a high-level language for writing constraint solvers. CHR are basically a committed-choice language consisting of guarded rules with multiple heads. There are two kinds of rules: Simplification rules rewrite constraints to simpler constraints while preserving logical equivalence (e.g. $X{\le}Y, Y{\le}X \Leftrightarrow X{=}Y$). Propagation rules add new constraints, which are logically redundant but may cause further simplification (e.g. $X{\le}Y, Y{\le}Z \Rightarrow X{\le}Z$). Repeated application of the rules incrementally solves constraints (e.g. $A{\le}B, B{\le}C, C{\le}A$ leads to $A{=}B,\ B{=}C$).
A simplification rule can be understood as a conditional term rewriting rule. Since a propagation rule does not rewrite constraints but adds new ones, conditional term rewriting systems cannot directly express them. Even though

every propagation rule (e.g. $\texttt{X}\leq\texttt{Y},\texttt{Y}\leq\texttt{Z} \Rightarrow \texttt{X}\leq\texttt{Z}$) can be written as a simplification rule (e.g. $\texttt{X}\leq\texttt{Y},\texttt{Y}\leq\texttt{Z} \Leftrightarrow \texttt{X}\leq\texttt{Y},\texttt{Y}\leq\texttt{Z},\texttt{X}\leq\texttt{Z}$), this is of little use, since such a simplification rule is applicable again and again. A propagation rule needs an "applicability condition" to prevent its reapplication. In implementations, termination of propagation rules is achieved by never applying a rule a second time to the same constraints. In this respect the operational semantics presented in [Frü95] is far from the implementation, since this applicability condition is not considered there (and thus a propagation rule can be applied infinitely many times). In this paper we give a new operational semantics that is more faithful to the actual implementations of CHR by avoiding the trivial nontermination of propagation rules.

Confluence

Typically, more than one rule is applicable to a conjunction of constraints. It is obviously desirable that the result of a computation in a solver will always be the same, semantically and syntactically, no matter which of the applicable rules is applied. This important property of any constraint solver is called confluence. In [AFM96] a decidable, sufficient and necessary syntactic condition for confluence of terminating simplification rules was introduced. This condition adopted and extended the terminology and techniques of conditional term rewriting systems [DOS88]. A straightforward translation of the results in this field was not possible, because the CHR formalism gives rise to phenomena not appearing in term rewriting systems. These phenomena include the existence of global knowledge (the built-in constraint store) and local variables (variables which appear only on the right-hand side of a rule).

The idea of the confluence criterion is to test joinability of finitely many minimal pairs of states (i.e. test whether the states result in the same final state). These so-called critical pairs can be derived from rules with overlapping heads (i.e. having at least a common instance of some head constraint). We then have to show that joinability of these minimal pairs is necessary and sufficient for joinability of arbitrary pairs of states, i.e. that critical pairs can be extended to any context in which two rules can be applied with different results.

Propagation rules are not covered by [AFM96]. An operational formulation of the semantics of the propagation rule turned out to be a bigger problem than it seems at a first glance. There are two conditions constraining this formulation: On the one hand we want to avoid trivial nontermination caused by applying the same propagation rule again and again. On the other hand the calculus defining the operational semantics should be monotonic in order to provide a well-defined system for the user and as a necessary condition for reasoning about confluence. Monotonicity means that if a computation can be executed in a context, then the same computation can be executed in any extension of this context (that contains additional information). In [Meu96] several applicability conditions for propagation rules were proposed and analyzed. The two main ideas for an applicability condition were integrating into the states a memory for all propagations rules together with the corresponding constraints that already fired (memory

condition) and a test, whether application of a propagation rule really adds new information to the state (redundancy test). But these applicability conditions failed to meet both requirements formulated above. The redundancy test does not avoid trivial nontermination. The memory condition results in a nonmonotonic calculus because the addition of information to the memory can inhibit the application of some propagation rules.
The study of applicability conditions in connection with confluence is necessary as the following example illustrates (the symbol `@` separates the rule name from the rule; see Section 2 for details of the syntax of CHR).

Example 1.1. Consider the following CHR program:

```
r1 @ p ⇒ q.
r2 @ r,q ⇔ true.
r3 @ r,p,q ⇔ s.
r4 @ s ⇔ p,q.
```

Rule `r1` is a propagation rule, while rules `r2,r3,r4` are simplification rules. Rule `r1` states that the constraint `p` implies `q`. Operationally, if we find `p` in the current state, we add the logical consequence `q` as redundant constraint. Rule `r2` means that the conjunction `r,q` is logically true. Rule `r3` says that the conjunction `r,p,q` can be simplified to `s`. Rule `r4` simplifies the constraint `s` into the conjunction `p,q`.

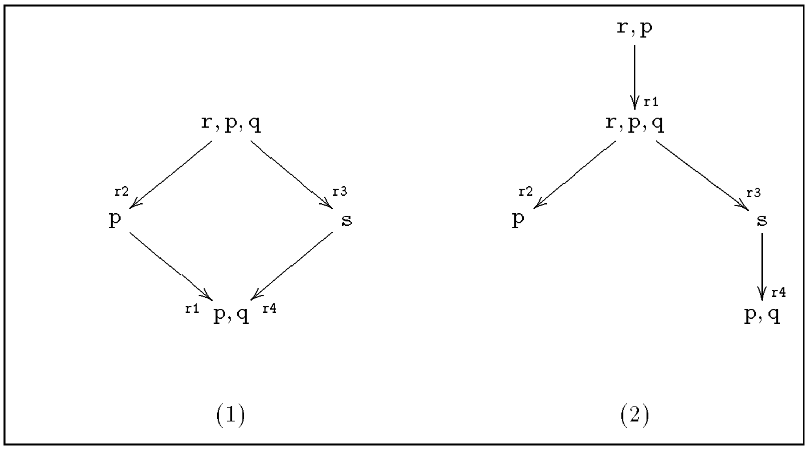

Figure 1. Violated Confluence

The rewriting of `r,p,q` leads in case (1) (Figure 1) to the same result `p,q` no matter which rules are applied in which order. In case (2) (Figure 1) the rewriting of r,p,q (which results from the application of r1 to r,p) leads to different results p and p,q since the propagation rule r1 has already been applied to p and thus cannot be reapplied. So confluence is violated.

Contribution of the paper

In this paper we refine the operational semantics presented in [Frü95] including a suitable applicability condition for propagation rules. To avoid the trivial non-termination of such rules our solution is to maintain information about propagation rules that can possibly be applied to a given set of user-defined constraints. This information consists of "tokens", which are a propagation rule and the set of candidate constraints. We integrate into the states a token store, and a propagation rule can only be applied if the token store contains the appropriate token. This token-based control leads to a monotonic operational semantics. In order to take into account the tokens we introduce then a new notion of critical pairs. Finally we give a decidable, sufficient and necessary syntactic condition for confluence of terminating CHR programs with propagation rules.

Organization of the Paper

The next section introduces the syntax of constraint handling rules (CHR) and the refined operational semantics. Section 3 extends the notion of confluence to CHR programs with propagation rules. Finally, we conclude with a summary and directions for future work.

2 Syntax and Operational Semantics of CHR

In this section we give the syntax and a new operational semantics of Constraint Handling Rules. We assume some familiarity with (concurrent) constraint logic programming [JM94, Sar93].
Constraint are considered to be special first-order predicates. We use two disjoint sorts of predicate symbols: *built-in predicates* and *user-defined predicates*. Intuitively, built-in predicates are defined by some constraint theory and handled by an appropriate constraint solver, while user-defined predicates are those defined by a CHR program. We call an atomic formula with a built-in predicate a *built-in constraint* and atomic formula with a user-defined predicate a *user-defined constraint*.
Throughout the paper we expect some constraint theory CT to be given which has the following properties:

- CT is consistent.
- CT does not contain any user-defined predicates.
- CT defines among other built-in predicates equality ("$\doteq$") as syntactic equality using for example Clark's axiomatization.

2.1 Syntax of CHR

Definition 2.1. A CHR *program* is a finite set of rules. There are two basic kinds of rules.[1] A *simplification rule* is of the form

$$\mathit{Rulename} \;@\; H_1, \ldots, H_i \Leftrightarrow G_1, \ldots, G_j \mid B_1, \ldots, B_k.$$

A *propagation rule* is of the form

$$\mathit{Rulename} \;@\; H_1, \ldots, H_i \Rightarrow G_1, \ldots, G_j \mid B_1, \ldots, B_k,$$

where *Rulename* is a unique identifier of a rule, the head $H_1, \ldots, H_i$ is a non-empty conjunction[2] of user-defined constraints, 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. Conjunctions of constraints as in the body are called *goals*. If the guard is empty, the symbol $\mid$ is omitted.

2.2 Operational Semantics of CHR

We define the operational semantics of a given CHR program P as a transition system that models the operations of the constraint solver defined by P.

States

Definition 2.2. A *state* is a tuple

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

Gs is a conjunction of user-defined and built-in constraints called *goal store*. C_U is a conjunction of user-defined constraints, likewise C_B is a conjunction of built-in constraints. C_U and C_B are called *user-defined and built-in (constraint) store*, respectively. $\mathcal{T}$ is a set of tokens (token store) of the form $R@C$, where C is a conjunction of user-defined constraints and R a rulename. $\mathcal{V}$ is an ordered sequence of variables. An empty goal store or user-defined store is represented by $\top$. The built-in store cannot be empty. In its most simple form it consists only of *true* or *false*.

Intuitively, Gs contains the constraints that remain to be solved, C_B and C_U are the built-in and the user-defined constraints, respectively, accumulated and simplified so far and $\mathcal{T}$ contains information about the propagation rules with the respective constraints which they can be possibly applied on.

[1] There is a third hybrid kind of rule called simpagation rule [BFL+94]. Since simpagation rules are abbreviations for simplification rules there is no need to discuss them in this paper.

[2] For conjunction in rules we use the symbol "," instead of "$\wedge$".

Definition 2.3. A variable X in a state $<Gs, C_U, C_B, \mathcal{T}, \mathcal{V}>$ is called

- *global*, if X appears in $\mathcal{V}$.
- *local*, if X does not appear in $\mathcal{V}$.
- *strictly local*, if X appears in C_B only.

Definition 2.4. The *logical reading* of a state $<Gs, C_U, C_B, \mathcal{T}, \mathcal{V}>$ is the formula

$$\exists \bar{y}\ Gs \wedge C_U \wedge C_B,$$

where $\bar{y}$ are the local variables of the state. Note that the global variables remain free in the formula.

Token store

A propagation rule needs an applicability condition to avoid trivial nontermination. Our approach stores information about propagation rules which can be possibly applied to a given set of user-defined constraints. Once a propagation rule has been applied to user-defined constraints, the appropriate token is removed (**Propagate**) so that the rule cannot be reapplied again to the same constraints.

The token store of a state contains rulenames together with the corresponding conjunctions of user-defined constraints that unify with the heads of the respective propagation rules. The updating of the token store by introducing user-defined constraints depends on the introduced constraints and the current user-defined store (**Introduce**).

This dependency is manifested in the multiset $T_{(C,C_U)}$ of tokens, which computes the new possibilities for apply propagation rules involving the new constraint C:

Definition 2.5. Let P be a CHR program and C a user-defined constraint.

$$\begin{aligned} T_{(C,C_U)} := \{R@H' \mid (R\ @\ H \Rightarrow G \mid B) \in P, H' &\text{ is a subconjunction of } C \wedge C_U, \\ C \text{ is a conjunct of } H', \text{ and } H' &\text{ unifies with } H\} \end{aligned}$$

is the *tokenset* of C with respect to C_U.

If C is a conjunction of constraints $C_1, \ldots, C_n$ then

$$T_{(C,C_U)} := T_{(C_1,C_U)} \cup T_{(C_2,C_1 \wedge C_U)} \cup \ldots \cup T_{(C_n,C_1 \wedge \ldots \wedge C_{n-1} \wedge C_U)}$$

Subconjunctions can be defined in the natural way similar to subsets:

Definition 2.6. Let $C = \bigwedge\limits_{i=1}^{n} C_i$ be a conjunction of constraints, π a permutation on $[1, \ldots, n]$, and $m \leq n$, then $\bigwedge\limits_{i=1}^{m} C_{\pi_k}$ is a subconjunction of C.

A Normal Form for States

We will assume that states are in a unique normal form that abstracts away the specifics of the built-in constraint solver: The normal form considers those states equivalent that impose the same built-in constraints and the same token store on the goal and on the user-defined constraint store. We model the normalization with a function that maps equivalent states into a syntactically unique representative state (up to variable renaming and order of conjuncts). The normalization function simplifies the built-in constraint store, projects out strictly local variables, propagates implied equations all over the state and deletes superfluous tokens from the token store. Most constraint solvers naturally support this functionality since they work with normal forms anyway. For the following theorems it is important to make the requirements on the normalization function more precise. The first three points of the definition of the normalization function $\mathcal{N}$ is a formalization of the update operation presented in [AFM96].

Definition 2.7. A function $\mathcal{N} : \mathcal{S} \to \mathcal{S}$, where $\mathcal{S}$ is the set of all states, is a *normalization function*, if it fulfills the following conditions:
Let $\mathcal{N}(<Gs, C_U, C_B, \mathcal{T}, \mathcal{V}>) = <Gs', C'_U, C'_B, \mathcal{T}', \mathcal{V}>$. We assume that there is a fixed order on all variables appearing in a state such that global variables are ordered as in $\mathcal{V}$ and precede all local variables.

1. *Equality propagation:* Gs', C'_U and $\mathcal{V}'$ derive from Gs, C_U and $\mathcal{V}$ by replacing all variables X, for which $CT \models \forall\,(C_B \to X \dot{=} t)$ holds,[3] by the corresponding term t, except if t is a variable that comes after X in the variable order.
2. *Projection:* The following must hold:
$$CT \models \forall\ ((\exists \bar{x} C_B) \leftrightarrow C'_B),$$
where $\bar{x}$ are the strictly local variables of $<Gs', C'_U, C_B, \mathcal{T}', \mathcal{V}>$.
3. *Uniqueness:* If
$$\begin{aligned} \mathcal{N}(<Gs_1, C_{U1}, C_{B1}, \mathcal{T}_1, \mathcal{V}>) &= <Gs'_1, C'_{U1}, C'_{B1}, \mathcal{T}'_1, \mathcal{V}> \text{ and} \\ \mathcal{N}(<Gs_2, C_{U2}, C_{B2}, \mathcal{T}_2, \mathcal{V}>) &= <Gs'_2, C'_{U2}, C'_{B2}, \mathcal{T}'_2, \mathcal{V}> \text{ and} \\ CT &\models (\exists \bar{x} C_{B1}) \leftrightarrow (\exists \bar{y} C_{B2}) \end{aligned}$$
holds, where $\bar{x}$ and $\bar{y}$, respectively, are the strictly local variables of the two states, then:
$$C'_{B1} = C'_{B2}.$$
4. *Token elimination:* $\mathcal{T}' = \mathcal{T} \cap T_{(C_U, \top)}$

The uniqueness property of $\mathcal{N}$ guarantees that there is exactly one representation for each set of equivalent built-in constraint stores. Therefore we can assume that an inconsistent built-in store is represented by the constraint *false*.

An important property of $\mathcal{N}$ is that it preserves the logical reading of states:

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

Lemma 2.8. Let be

$$\mathcal{N}(<Gs, C_U, C_B, \mathcal{T}, \mathcal{V}>) = <Gs', C'_U, C'_B, \mathcal{T}', \mathcal{V}>.$$

Then the following equivalence holds

$$CT \models \forall\ (\exists \bar{x}(Gs \wedge C_U \wedge C_B) \leftrightarrow \exists \bar{x}'(Gs' \wedge C'_U \wedge C'_B)),$$

where $\bar{x}$ and $\bar{x}'$ are the local variables in S and S', respectively.

Computation Steps

Given a CHR program P we define the transition relation $\mapsto$ by introducing four kinds of *computation steps*. The aim of the computation is to incrementally reduce arbitrary states to states that contain no goals and, if possible, the simplest form of user-defined constraints.

Notation: Capital letters denote conjunctions of constraints. By equating two constraints, $c(t_1, \ldots, t_n) \doteq c(s_1, \ldots, s_n)$, we mean $t_1 \doteq s_1 \wedge \ldots \wedge t_n \doteq s_n$. By $(p_1 \wedge \ldots \wedge p_n) \doteq (q_1 \wedge \ldots \wedge q_n)$ we mean $p_1 \doteq q_1 \wedge \ldots \wedge p_n \doteq q_n$. Note that conjuncts can be permuted since conjunction is associative and commutative, and that we will identify all states containing the built-in store *false*.

Solve

$$\frac{C \text{ is a built-in constraint}}{<C \wedge Gs, C_U, C_B, \mathcal{T}, \mathcal{V}> \mapsto \mathcal{N}(<Gs, C_U, C \wedge C_B, \mathcal{T}, \mathcal{V}>)}$$

Introduce

$$\frac{C \text{ is a user-defined constraint}}{<C \wedge Gs, C_U, C_B, \mathcal{T}, \mathcal{V}> \mapsto \mathcal{N}(<Gs, C \wedge C_U, C_B, \mathcal{T} \cup T_{(C, C_U)}, \mathcal{V}>)}$$

Simplify

$$\frac{\begin{array}{c}(H \Leftrightarrow G \mid B) \text{ is a fresh variant of a rule in } P \text{ with the variables } \bar{x} \\ CT \models C_B \rightarrow \exists \bar{x}(H \doteq H' \wedge G)\end{array}}{<Gs, H' \wedge C_U, C_B, \mathcal{T}, \mathcal{V}> \mapsto \mathcal{N}(<Gs \wedge B, C_U, H \doteq H' \wedge C_B, \mathcal{T}, \mathcal{V}>)}$$

Propagate

$$\frac{\begin{array}{c}(R@H \Rightarrow G \mid B) \text{ is a fresh variant of a rule in } P \text{ with the variables } \bar{x} \\ CT \models C_B \rightarrow \exists \bar{x}(H \doteq H' \wedge G)\end{array}}{<Gs, H' \wedge C_U, C_B, \{R@H'\} \cup \mathcal{T}, \mathcal{V}> \mapsto \mathcal{N}(<Gs \wedge B, H' \wedge C_U, H \doteq H' \wedge C_B, \mathcal{T}, \mathcal{V}>)}$$

Figure 2. Computation Steps

In the **Solve** computation step, the built-in solver normalizes the resulting state after moving a constraint C from the goal store to the built-in store. **Introduce** transports a user-defined constraint C from the goal store into the user-defined constraint store and adds the tokenset of C with respect to C_U, $T_{(C,C_U)}$ (i.e. information about the propagation rules that can fire after adding a new constraint to the user-defined store), to the token store, and finally normalizes the resulting state. To **Simplify** user-defined constraints H' means to replace them by the body B of a fresh variant[4] of a simplification rule $(H \Leftrightarrow G \mid B)$ from the program, provided H' matches[5] the head H and the resulting guard G is implied by the built-in constraint store, and finally to normalize the resulting state. To **Propagate** user-defined constraints H' means to add B to the goal store Gs and remove the token $R@H$ from the token store if H' matches the head H of a propagation rule $(H \Rightarrow G \mid B)$ in the program and the resulting guard G is implied by the built-in constraint store, and finally to normalize the resulting state.

Lemma 2.9. Normalization has no influence on application of rules, i.e.

$$S \mapsto S' \text{ holds iff } \mathcal{N}(S) \mapsto S'.$$

This claim is shown by analyzing each kind of computation step.

Definition 2.10. $S \mapsto^* S'$ holds iff

$$S = S' \text{ or } S' = \mathcal{N}(S) \text{ or } S \mapsto S_1 \mapsto \ldots \mapsto S_n \mapsto S' \quad (n \geq 0).$$

Definition 2.11. An *initial state* for a goal Gs is of the form:

$<Gs, \top, \mathrm{true}, \emptyset, \mathcal{V}>$,

where $\mathcal{V}$ is the sequence of the variables occuring in Gs.
A *final state* is either of the form

$<Gs, C_U, \mathrm{false}, \mathcal{T}, \mathcal{V}>$

(such a state is called *failed*) or of the form

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

with no computation step possible anymore and C_B not *false* (such a state is called *successful*).

Example 2.12. We define a user-defined constraint for less-than-or-equal, $\leq$, that can handle variable arguments:

```
r1 @ X≤X ⇔  true.
r2 @ X≤Y, Y≤X ⇔ X=Y.
r3 @ X≤Y, Y≤Z ⇒ X≤Z.
r4 @ X≤Y, X≤Y ⇔ X≤Y.
```

[4] Two expressions are variants if they can be obtained from each other by a variable renaming. A fresh variant contains only variables that do not occur in the state.

[5] Matching rather than unification is the effect of the existential quantification over the head equalities: $\exists \bar{x}(H \doteq H')$.

The CHR program implements reflexivity (`r1`), antisymmetry (`r2`), transitivity (`r3`) and idempotence (`r4`) in a straightforward way. The reflexivity rule `r1` states that $\mathtt{X}\leq\mathtt{X}$ is logically true. Hence, whenever we see the constraint $\mathtt{X}\leq\mathtt{X}$ we can simplify it to `true`. The antisymmetry rule `r2` means that if we find $\mathtt{X}\leq\mathtt{Y}$ as well as $\mathtt{Y}\leq\mathtt{X}$ in the current store, we can replace them by the logically equivalent `X=Y`. The transitivity rule `r3` propagates constraints. It states that the conjunction $\mathtt{X}\leq\mathtt{Y},\ \mathtt{Y}\leq\mathtt{Z}$ implies $\mathtt{X}\leq\mathtt{Z}$. Operationally, we add the logical consequence $\mathtt{X}\leq\mathtt{Z}$ as a redundant constraint. The idempotence rule `r4` absorbs multiple occurrences of the same constraint.
A computation of the goal $\mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A} \wedge \mathtt{B} \leq \mathtt{C}$ proceeds as follows (Note that $\mathcal{V} = [\mathtt{A}, \mathtt{B}, \mathtt{C}]$[6]):

$$
\begin{array}{ll}
 & <\underline{\mathtt{A} \leq \mathtt{B}} \wedge \mathtt{C} \leq \mathtt{A} \wedge \mathtt{B} \leq \mathtt{C}, \top, \mathtt{true}, \emptyset, \mathcal{V}> \\
\mapsto (\textbf{Introduce}) & <\underline{\mathtt{C} \leq \mathtt{A}} \wedge \mathtt{B} \leq \mathtt{C}, \mathtt{A} \leq \mathtt{B}, \mathtt{true}, \emptyset, \mathcal{V}> \\
\mapsto (\textbf{Introduce}) & <\underline{\mathtt{B} \leq \mathtt{C}}, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A}, \mathtt{true}, \mathcal{T}_1, \mathcal{V}> \\
 & \mathcal{T}_1 = \{\mathtt{r3@C} \leq \mathtt{A} \wedge \mathtt{A} \leq \mathtt{B}\} \\
\mapsto (\textbf{Introduce}) & <\top, \underline{\mathtt{A} \leq \mathtt{B}} \wedge \underline{\mathtt{C} \leq \mathtt{A}} \wedge \mathtt{B} \leq \mathtt{C}, \mathtt{true}, \mathcal{T}_2, \mathcal{V}> \\
 & \mathcal{T}_2 = \mathcal{T}_1 \cup \{\mathtt{r3@A} \leq \mathtt{B} \wedge \mathtt{B} \leq \mathtt{C}, \mathtt{r3@B} \leq \mathtt{C} \wedge \mathtt{C} \leq \mathtt{A}\} \\
\mapsto (\textbf{Propagate} \text{ with } \mathtt{r3}) & <\underline{\mathtt{C} \leq \mathtt{B}}, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A} \wedge \mathtt{B} \leq \mathtt{C}, \mathtt{true}, \mathcal{T}_3, \mathcal{V}> \\
 & \mathcal{T}_3 = \{\mathtt{r3@A} \leq \mathtt{B} \wedge \mathtt{B} \leq \mathtt{C}, \mathtt{r3@B} \leq \mathtt{C} \wedge \mathtt{C} \leq \mathtt{A}\} \\
\mapsto (\textbf{Introduce}) & <\top, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A} \wedge \underline{\mathtt{B} \leq \mathtt{C}} \wedge \underline{\mathtt{C} \leq \mathtt{B}}, \mathtt{true}, \mathcal{T}_4, \mathcal{V}> \\
 & \mathcal{T}_4 = \mathcal{T}_3 \cup \{\mathtt{r3@C} \leq \mathtt{B} \wedge \mathtt{B} \leq \mathtt{C}, \mathtt{r3@B} \leq \mathtt{C} \wedge \mathtt{C} \leq \mathtt{B}\} \\
\mapsto (\textbf{Simplify} \text{ with } \mathtt{r2}) & <\underline{\mathtt{B} = \mathtt{C}}, \mathtt{A} \leq \mathtt{B} \wedge \mathtt{C} \leq \mathtt{A}, \mathtt{true}, \mathcal{T}_5, \mathcal{V}> \\
 & \mathcal{T}_5 = \{\mathtt{r3@C} \leq \mathtt{A} \wedge \mathtt{A} \leq \mathtt{B}\} \\
\mapsto (\textbf{Solve}) & <\top, \underline{\mathtt{A} \leq \mathtt{B}} \wedge \underline{\mathtt{B} \leq \mathtt{A}}, \mathtt{B} = \mathtt{C}, \mathcal{T}_6, \mathcal{V}> \\
 & \mathcal{T}_6 = \{\mathtt{r3@B} \leq \mathtt{A} \wedge \mathtt{A} \leq \mathtt{B}\} \\
\mapsto (\textbf{Simplify} \text{ with } \mathtt{r2}) & <\underline{\mathtt{A} = \mathtt{B}}, \top, \mathtt{B} = \mathtt{C}, \emptyset, \mathcal{V}> \\
\mapsto (\textbf{Solve}) & <\top, \top, \mathtt{A} = \mathtt{B} \wedge \mathtt{B} = \mathtt{C}, \emptyset, \mathcal{V}>
\end{array}
$$

Note that, since the application of **Simplify** removes constraints from the user-defined store, $\mathcal{N}$ reduces the set of tokens accordingly.

3 Confluence of CHR programs

We adopt and extend the terminology and techniques of conditional term rewriting systems [DOS88]. A straightforward translation of the results in this field was not possible, because the CHR formalism gives rise to phenomena not appearing in term rewriting systems. Similar to contextual rewriting [ZR85], CHR programs are more powerful than the classical conditional rewriting because they use an additional context. On the one hand the entailment test requires knowledge of the current state of the built-in store. On the other hand the application of a propagation rule requires knowledge of the current state of the token store. Confluence guarantees that any computation starting from an arbitrary given initial state results in the same final state. We first define what it means that

[6] In the following the square brackets [ ] denote a sequence.

two computations have the same result.

Definition 3.1. 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' and S_2' are variants.

Definition 3.2. 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 3.3. A CHR program is called *locally confluent* if the following holds for all states S, S_1, S_2:

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

The joinability of all state pairs derived from a common direct ancestor state can not be checked to analyze the local confluence of a given CHR program due to the existence of infinitely such states. However, it is possible to construct a finite number of minimal states where more than one rule is applicable. A direct common ancestor state consists of the heads and guards of the rules. It is obvious that there is only a finite number of such states for a given program. These states can be extended to any context, i.e. to all possible ancestor states by adding constraints and tokens to the components of the state.
We now further restrict to nontrivial direct common ancestor states: Joinability can only be destroyed if one rule inhibits the application of the other rule. The application of a rule may remove constraints from the user-defined store and introduce new constraints. Only the removal of constraints can affect the applicability of another rule, in case the removed constraint is needed by the other rule. To possibly inhibit each other, one rule must be a simplification rule and the two rules must overlap, i.e. have at least one head atom in common in the ancestor state. The pair of states resulting from this overlap is called *critical pair*.

Definition 3.4. Let R be a rule.

$$\text{survive}(R) := \begin{cases} \quad\text{true} \quad, & \text{if } R \text{ is a simplification rule} \\ H_1 \wedge \ldots \wedge H_n, & \text{if } R \text{ is a propagation rule with head } H_1, \ldots, H_n \end{cases}$$

survive(R) computes the conjunction of those constraints from the head of a rule R that will not be deleted from the constraint store by the application of R.

Definition 3.5. Let R be a rule with guard G, body B and head H and let R' be a rule with guard G', body B' and head H'. R and R' are simplification rules or a simplification rule and a propagation one. Let $\{H_i \,|\, 1 \leq i \leq n\}$ and $\{H_i' \,|\, 1 \leq i \leq m\}$ be the set of atoms H and H' respectively, then the triple

$$\left(\begin{array}{c} G \wedge G' \wedge H_{i_1} \doteq H'_{j_1} \wedge \ldots \wedge H_{i_k} \doteq H'_{j_k}\,, \\ (B, \text{survive}(R) \wedge H'_{j_{k+1}} \wedge \ldots \wedge H'_{j_m}) =\downarrow= (B', \text{survive}(R') \wedge H_{i_{k+1}} \wedge \ldots \wedge H_{i_n})\,, \\ \mathcal{V} \end{array} \right)$$

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

Example 3.6. Consider the program for $\le$ of Example 2.12. The following critical pair stems from unifying the first atom of the head of the antisymmetry rule (r2) with the first atom of the head of the transitivity rule (r3):

$(\texttt{true} \quad , \quad (\texttt{X}\le\texttt{Z}, \texttt{X}\le\texttt{Y}\land\texttt{Y}\le\texttt{Z}\land\texttt{Y}\le\texttt{X}) = \downarrow = (\texttt{X=Y}, \texttt{Y}\le\texttt{Z}) \quad , \quad [\texttt{X,Y,Z}])$

Critical pairs represent minimal pairs of states resulting from an overlap. Since critical pairs are minimal, the token store of the states must also be minimal. Two rules can only be applied to the overlap if the user-defined store contains the appropriate user-defined constraints and the token store contains the appropriate token, in case the applied rule is a propagation rule. After applying the rules to the overlap the user-defined stores of the resulting states contain the remaining user-defined constraints, the goal stores will be extended with the constraints coming from the body of the rules, and the appropriate token is removed from the token stores either by the transition **Propagate** or by the normalization function $\mathcal{N}$, in case the applied rule is a simplification rule (i.e., the removed constraints occur in the appropriate token).

Definition 3.7. A critical pair $(G\,,\,(B_1, H_1) =\downarrow= (B_2, H_2)\,,\,\mathcal{V})$ is called *joinable* if $<B_1, H_1, G, \emptyset, \mathcal{V}>$ and $<B_2, H_2, G, \emptyset, \mathcal{V}>$ are joinable.

The choice of the token stores of the states represented by the critical pair is motivated by the minimality criterion for these states: it covers the case that all propagation rules (except possibly one) have already been applied to the constraints of the user-defined store before the direct ancestor state was reached.

Example 3.8. The critical pair in Example 3.6 is joinable. A computation sequence beginning with $<\texttt{X} \le \texttt{Z}, \texttt{X} \le \texttt{Y} \land \texttt{Y} \le \texttt{Z} \land \texttt{Y} \le \texttt{X}, \texttt{true}, \emptyset, \mathcal{V}>$, where $\mathcal{V} = [\texttt{X}, \texttt{Y}, \texttt{Z}]$ proceeds as follows:

$$
\begin{array}{ll}
 & <\underline{\texttt{X} \le \texttt{Z}}, \texttt{X} \le \texttt{Y} \land \texttt{Y} \le \texttt{Z} \land \texttt{Y} \le \texttt{X}, \texttt{true}, \emptyset, \mathcal{V}> \\
\mapsto (\textbf{Introduce}) & <\top, \texttt{X} \le \texttt{Z} \land \underline{\texttt{X} \le \texttt{Y}} \land \texttt{Y} \le \texttt{Z} \land \underline{\texttt{Y} \le \texttt{X}}, \texttt{true}, \mathcal{T}, \mathcal{V}> \\
 & \mathcal{T} = \{\texttt{r3@Y} \le \texttt{X} \land \texttt{X} \le \texttt{Z}\} \\
\mapsto (\textbf{Simplify}\text{ with }\texttt{r2}) & <\underline{\texttt{X} = \texttt{Y}}, \texttt{X} \le \texttt{Z} \land \texttt{Y} \le \texttt{Z}, \texttt{true}, \emptyset, \mathcal{V}> \\
\mapsto (\textbf{Solve}) & <\top, \underline{\texttt{X} \le \texttt{Z}} \land \underline{\texttt{X} \le \texttt{Z}}, \texttt{X} = \texttt{Y}, \emptyset, \mathcal{V}> \\
\mapsto (\textbf{Simplify}\text{ with }\texttt{r4}) & <\top, \texttt{X} \le \texttt{Z}, \texttt{X} = \texttt{Y}, \emptyset, \mathcal{V}>
\end{array}
$$

A computation sequence beginning with $<\texttt{X} = \texttt{Y}, \texttt{Y} \le \texttt{Z}, \texttt{true}, \emptyset, \mathcal{V}>$ results in the same final state:

$$
\begin{array}{l}
\qquad <\underline{\texttt{X} = \texttt{Y}}, \texttt{Y} \le \texttt{Z}, \texttt{true}, \emptyset, \mathcal{V}> \\
\mapsto (\textbf{Solve}) <\top, \texttt{X} \le \texttt{Z}, \texttt{X} = \texttt{Y}, \emptyset, \mathcal{V}>
\end{array}
$$

With this new notion of critical pairs we are now in a position to give a sufficient and necessary condition for local confluence. The idea of this criterion is to test joinability of the critical pairs. We then have to show that joinability of these minimal pairs is necessary and sufficient for joinability of arbitrary pairs of states with a common direct ancestor, i.e. that critical pairs can be extended to any context. The proof for the following theorem can be found in [Abd97]. The proof is an extension and simplification of the one presented in [AFM96] taking into account the propagation rules and their tokens.

Theorem 3.9. A CHR program is locally confluent iff all its critical pairs are joinable.

Proof. (Idea) *The if-direction:* Assume that we are in state S where there are two or more possibilities of computation:

$$S \mapsto S_1 \text{ and } S \mapsto S_2.$$

We investigate all pairs of possible computation steps and show that S_1 and S_2 are joinable. There are ten relevant combinations:

1. **Solve + Solve**
2. **Solve + Introduce**
3. **Solve + Simplify**
4. **Solve + Propagate**
5. **Introduce + Introduce**
6. **Introduce + Simplify**
7. **Introduce + Propagate**
8. **Simplify + Simplify**
9. **Simplify + Propagate**
10. **Propagate + Propagate**

According to the assumption that logical consequence is monotonous and that the constraint theory preserves commutativity of the conjunction, 1-7 are easily shown. The cases 8 and 9 are the main part of the proof: Similar to conditional term rewriting systems [DOS88] we distinguish two situations: Disjoint Peaks (i.e., no constraint of the head of the rule unifies with a constraint of the head of the other rule) and Critical Peaks (i.e., at least one constraint of the head of the rule unifies with a constraint of the head of the other rule). The first situation is trivial. For the proof of the second situation the assumption that all critical pairs are joinable is needed. We show that critical pairs can be extended to S_1 and S_2 without losing joinability. Case 10 is also easily shown, since the application of two propagation rules onto a state requires that the token store of this state contains the appropriate tokens and that after application of **Propagate** only one token will be removed.

The only-if-direction: We assume that we have a locally confluent CHR program with a critical pair, which is not joinable. We lead this assumption to a contradiction. We distinguish two different cases: The non-joinable critical pair either stems from two simplification rules or stems from a simplification rule and a propagation rule. We construct a state, on which the application of the rules leads to the states represented by the non-joinable critical pair. Since the program is locally confluent the states must be joinable. This leads to a contradiction.

Definition 3.10. A CHR program is called *terminating*, if there are no infinite computations.

The following corollary is an immediate consequence of Theorem 3.9 and Newman's lemma [New42]:

Corollary 3.11. A terminating CHR program is confluent iff its critical pairs are joinable.

The Corollary 3.11 gives a decidable characterization of confluent terminating CHR programs: Joinability of a given critical pair is decidable for a terminating CHR program (i.e. finite computations) and there are only finitely many critical pairs.
The CHR program presented in Example 1.1 is not confluent. We show now, how our token-based approach can detect the non-confluence.

Example 3.12. We consider one of the critical pairs stemming from the rules `r2` and `r3` of Example 1.1:
The critical pair `(true , (true,p)=`$\downarrow$`=(s,`$\top$`) , [])` is not joinable, because computation sequences beginning with the states $<\mathtt{true}, \mathtt{p}, \mathtt{true}, \emptyset, \emptyset>$ and $<\mathtt{s}, \top, \mathtt{true}, \emptyset, \emptyset>$ do not result in the same final state:

$$
\begin{array}{ll}
 & <\underline{\mathtt{true}}, \mathtt{p}, \mathtt{true}, \emptyset, \emptyset> \\
\mapsto (\textbf{Solve}) & <\top, \mathtt{p}, \mathtt{true}, \emptyset, \emptyset>
\end{array}
$$

All computation sequences beginning with $<\mathtt{s}, \top, \mathtt{true}, \emptyset, \emptyset>$ will result in the same state even if they differ in the order of application of computation steps:

$$
\begin{array}{ll}
 & <\underline{\mathtt{s}}, \top, \mathtt{true}, \emptyset, \emptyset> \\
\mapsto (\textbf{Introduce}) & <\top, \underline{\mathtt{s}}, \mathtt{true}, \emptyset, \emptyset> \\
\mapsto (\textbf{Simplify} \text{ with } \mathtt{r4}) & <\underline{\mathtt{p}} \wedge \mathtt{q}, \top, \mathtt{true}, \emptyset, \emptyset> \\
\mapsto (\textbf{Introduce}) & <\underline{\mathtt{q}}, \mathtt{p}, \mathtt{true}, \{\mathtt{r1@p}\}, \emptyset> \\
\mapsto (\textbf{Introduce}) & <\top, \underline{\mathtt{p}} \wedge \mathtt{q}, \mathtt{true}, \{\mathtt{r1@p}\}, \emptyset> \\
\mapsto (\textbf{Propagate} \text{ with } \mathtt{r1}) & <\underline{\mathtt{q}}, \mathtt{p} \wedge \mathtt{q}, \mathtt{true}, \emptyset, \emptyset> \\
\mapsto (\textbf{Introduce}) & <\top, \mathtt{p} \wedge \mathtt{q} \wedge \mathtt{q}, \mathtt{true}, \emptyset, \emptyset>
\end{array}
$$

4 Conclusion and Future Work

In this paper, we have presented for the first time a refined operational semantics for Constraint Handling Rules that is more faithful to their actual implementations. This operational semantics avoids the trivial nontermination of the propagation rules. Our approach was to maintain information about propagation rules that can possibly be applied to a given set of user-defined constraints in the form of tokens. A propagation rule can only be applied if the token store contains the appropriate token.

We solved the open problem of a confluence test for CHR programs with propagation rules by extending the notion of critical pairs taking into account the token-based control and giving a decidable, sufficient and necessary condition for confluence through joinability of critical pairs (extending the results of [AFM96]). Interesting directions for future work include

- investigation of so-called completion methods to make a non-confluent CHR program confluent and
- determination of sufficient conditions guaranteeing that the combination of confluent constraint solvers is confluent as well.

Acknowledgements

I would like to thank Thom Frühwirth, Holger Meuss and Norbert Eisinger for useful comments on a preliminary version of this paper.

References

Abd97. S. Abdennadher. Rewriting concepts in the study of confluence of constraint handling rules. Technical report PMS-FB-1997-15, Institute of Computer Science, Ludwig–Maximilians–University Munich, January 1997.

AFM96. S. Abdennadher, T. Frühwirth, and H. Meuss. On confluence of constraint handling rules. In E. Freuder, editor, *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, CP'96*, LNCS 1118. Springer, August 1996.

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

DOS88. N. Dershowitz, N. Okada, and G. Sivakumar. Confluence of conditional rewrite systems. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings of the 1st International Workshop on Conditional Term Rewriting Systems*, LNCS 308, pages 31–44, 1988.

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

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

Meu96. Holger Meuss. Konfluenz von Constraint-Handling-Rules-Programmen. Master's thesis, Institut für Informatik, Ludwig–Maximilians–Universität München, 1996.

New42. M. H. A. Newman. On theories with a combinatorial definition of equivalence. In *Annals of Math*, volume 43, pages 223–243, 1942.

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

vH91. P. van Hentenryck. Constraint logic programming. *The Knowledge Engineering Review*, 6:151–194, 1991.

ZR85. H. Zhang and J. L. Remy. Contextual rewriting. In Jean-Pierre Jouannaud, editor, *Proceedings of the 1st International Conference on Rewriting Techniques and Applications*, volume 202 of *LNCS*, pages 46–62, Dijon, France, May 1985. Springer.