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

Ludwig-
Maximilians-
Universität
München

Operational Equivalence of CHR Programs And Constraints

Slim Abdennadher and Thom Frühwirth

Fifth International Conference on Principles and Practice of Constraint Programming, CP'99, Alexandra, Virginia, USA
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1999-9, August 1999

Operational Equivalence of CHR Programs And Constraints

Slim Abdennadher and Thom Frühwirth

Computer Science Department, University of Munich
Oettingenstr. 67, D-80538 Munich, Germany
{Slim.Abdennadher, Thom.Fruehwirth}@informatik.uni-muenchen.de

Abstract. A fundamental question in programming language semantics is when two programs should be considered equivalent. In this paper we introduce a notion of operational equivalence for CHR programs and user-defined constraints. Constraint Handling Rules (CHR) is a high-level language for writing constraint solvers either from scratch or by modifying existing solvers.
We give a decidable, sufficient and necessary syntactic condition for operational equivalence of terminating and confluent CHR programs.
For practical reasons, we also investigate a notion of operational equivalence for user-defined constraints that are defined in different programs. We give a sufficient syntactic condition for constraints defined in terminating and confluent CHR programs. For a subclass of programs which have only one user-defined constraint in common, we are able to give a sufficient and necessary syntactic condition.

1 Introduction

Constraint Handling Rules (CHR) [Frü98] is essentially a committed-choice language consisting of multi-headed guarded rules that transform constraints into simpler ones until they are solved. CHR defines both *simplification* of and *propagation* over user-defined constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence, e.g. $\mathtt{X}\geq\mathtt{Y}\wedge\mathtt{Y}\geq\mathtt{X} \Leftrightarrow \mathtt{X}=\mathtt{Y}$. Propagation adds new constraints, which are logically redundant but may cause further simplification, e.g. $\mathtt{X}\geq\mathtt{Y}\wedge\mathtt{Y}\geq\mathtt{Z} \Rightarrow \mathtt{X}\geq\mathtt{Z}$.
As a special-purpose language for writing constraint solvers, CHR aims to fulfill the promise of user-defined constraints as stated in [ACM96]: "For the theoretician meta-theorems can be proved and analysis techniques invented once and for all; for the implementor different constructs (backward and forward chaining, suspension, compiler optimization, debugging) can be implemented once and for all; for the user only one set of ideas need to be understood, though with rich (albeit disciplined) variations (constraint systems)."

A fundamental and hard question in programming language semantics is when two programs should be considered equivalent. For example correctness of program transformation can be studied only with respect to a notion of equivalence.

Also, if modules or libraries with similar functionality are used together, one may be interested in finding out if program parts in different modules or libraries are equivalent. In the context of CHR, this case arises frequently when constraint solvers written in CHR are combined. Typically, a constraint is only partially defined in a constraint solver. We want to make sure that the operational semantics of the common constraints of two programs do not differ, and we are interested in finding out if they are equivalent.
The literature on equivalence of programs in logic-based languages is sparse. In most papers that touch the subject, a suitable notion of program equivalence serves as a correctness criterion for transformations between programs, e.g. in partial evaluation and deduction. Our concern is the problem of program equivalence in its generality, where the programs to be compared are independent from each other.
[Mah86] provides a systematic comparison of the relative strengths of various formulations of equivalence of logic programs. These formulations arise naturally from several formal semantics of logic programs. Maher does not study how to test for equivalence. The results may be extensible to constraint logic programs, but committed-choice languages like CHR have different semantics that induce different notions of equivalence. In particular, in CHR the distinction between successful, failed or deadlocked goals is secondary, but the distinction between a goal and its instances is vital. For similar reasons, [GLM95] among other things extends Maher's work by considering relationships between equivalences derived from semantics that are based e.g. on computed answer substitutions. Gabbrielli et. al. are not concerned with tests for equivalence, either.
Like [GLM95] we are concerned with equivalences of the observable behavior of programs. Observables are then a suitable abstraction of execution traces. In case of equivalence based on operational semantics expressed by a transition system, it is common to define as observables the results of finite computations, where one abstracts away local variables, see e.g. [EGM98].

We have already shown in previous work [Abd97] that analysis techniques are available for an important property of any constraint solver, namely confluence: The result of a computation should be independent from the order in which constraints arrive and in which rules are applied to the constraints. For confluence of terminating CHR programs we were able to give a decidable, sufficient and necessary condition [Abd97,AFM99]. A completion algorithm that makes programs confluent if it terminates, was presented in [AF98].
It is tempting to think that a suitable modification of the concept of confluence can be used to express equivalence of programs. In this paper we show that a straightforward application of our confluence test is too weak to capture the operational equivalence of CHR programs.
In practice, one is often interested in comparing implementations of constraints instead of whole programs. Hence we investigate a notion of operational equivalence for user-defined constraints that are defined in different programs. We give a sufficient syntactic condition for constraints defined in terminating and confluent CHR programs. For a subclass of programs which have only one user-defined

constraint in common, we are able to give a sufficient and necessary syntactic condition.
Based on these results, we are finally able to give a decidable, sufficient and necessary syntactic condition for operational equivalence of terminating and confluent CHR programs.

This paper is organized as follows: In Section 2 we define the CHR language and summarize previous confluence results. Section 3 presents our notion of operational equivalence for CHR and the results about this notion. Finally, we conclude with a summary and directions for future work.

2 Preliminaries

In this section we give an overview of syntax and semantics as well as confluence results for constraint handling rules. More detailed presentations can be found in [AFM96,Abd97,AFM99].

2.1 Syntax of CHR

We use two disjoint kinds of predicate symbols for two different classes of constraints: *built-in constraint symbols* and *user-defined constraint symbols (CHR symbols)*. We call an atomic formula with a constraint symbol a *constraint*. Built-in constraints are handled by a predefined, given constraint solver that already exists as a certified black-box solver. user-defined constraints are defined in a CHR program.
A *CHR program* is a finite set of rules. There are two kinds of rules:
A *simplification rule* is of the form

$$H \Leftrightarrow C \mid B.$$

A *propagation rule* is of the form

$$H \Rightarrow C \mid B,$$

where the *head* H is a non-empty conjunction of user-defined constraints, the *guard* C is a conjunction of built-in constraints, and the *body* B is a goal. A *goal* is a conjunction of built-in and user-defined constraints. A guard "*true*" is usually omitted together with the commit symbol $\mid$. A CHR symbol is *defined* in a CHR program if it occurs in the head of a rule in the program.

2.2 Declarative Semantics of CHR

The logical meaning of a simplification rule is a logical equivalence provided the guard holds. $\forall F$ denotes the universal closure of a formula F.

$$\forall(C \rightarrow (H \leftrightarrow \exists \bar{y}\ B)),$$

where $\bar{y}$ are the variables that appear only in the body B.
The logical meaning of a propagation rule is an implication provided the guard holds

$$\forall(C \rightarrow (H \rightarrow \exists\bar{y}\ B)).$$

The logical meaning $\mathcal{P}$ of a CHR program P is the conjunction of the logical meanings of its rules united with a consistent *constraint theory* CT that defines the built-in constraint symbols. We require CT to define the constraint symbol $=$ as syntactic equality.

2.3 Operational Semantics of CHR

The operational semantics of CHR is given by a transition system.
A state $G_{\mathcal{V}}$ is a goal G together with a sequence of variables, $\mathcal{V}$. Where it is clear from the context, we will drop the annotation $\mathcal{V}$.
We require that states are normalized so that they can be compared syntactically in a meaningful way. Basically, we require that the built-in constraints are in a (unique) normal form, where all syntactic equalities are made explicit and are propagated to the user-defined constraints. Furthermore, we require that the normalization projects out strictly local variables, i.e. variables appearing in the built-in constraints only. A precise definition of the normalization function $\mathcal{N}$ can be found in [Abd97,AFM99].
Given a CHR program P we define the transition relation $\mapsto_P$ by introducing two kinds of computation steps (Figure 1). $\mapsto_P^+$ denotes the transitive closure, $\mapsto_P^*$ denotes the reflexive and transitive closure of $\mapsto_P$.
An *initial state* for a goal G is the state $\mathcal{N}(G_{\mathcal{V}})$, where $\mathcal{N}$ is a function that normalizes a state as defined below and $\mathcal{V}$ is a sequence of all variables appearing in G. A *final state* is one where either no computation step is possible anymore or where the built-in constraints are inconsistent.
A *computation* of a goal G in a program P is a sequence $S_0, S_1, \ldots$ of states with $S_i \mapsto_P S_{i+1}$ beginning with the initial state for G and ending in a final state or diverging. Where it is clear from the context, we will drop the reference to the program P.
In Figure 1, the notation G_{built} denotes the built-in constraints in a goal G. We will also use the notation G_{user} to denote the user-defined constraints in a goal G. An equation $c(t_1, \ldots, t_n){=}d(s_1, \ldots, s_n)$ of two constraints stands for $t_1{=}s_1 \land \ldots \land t_n{=}s_n$ if c and d are the same CHR symbols and for *false* otherwise. An equation $(p_1 \land \ldots \land p_n){=}(q_1 \land \ldots \land q_m)$ stands for $p_1{=}q_1 \land \ldots \land p_n{=}q_n$ if $n = m$ and for *false* otherwise. Conjuncts can be permuted since conjunction is associative and commutative.
To **Simplify** user-defined constraints H' means to remove them from the state $H' \land G$ and to add the body B of a fresh variant of a simplification rule $(H \Leftrightarrow C \mid B)$ and the equation $H{=}H'$ and the guard C to the resulting state G, provided H' matches the head H and the guard C is implied by the built-in constraints appearing in G, and finally to normalize the resulting state. In this case we say

Simplify

If	$(H \Leftrightarrow C \mid B)$ is a fresh variant of a rule with variables $\bar{x}$
and	$CT \models \forall\,(G_{built} \rightarrow \exists\bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G)_{\mathcal{V}} \mapsto_P \mathcal{N}((H{=}H' \wedge B \wedge C \wedge G)_{\mathcal{V}})$

Propagate

If	$(H \Rightarrow C \mid B)$ is a fresh variant of a rule with variables $\bar{x}$
and	$CT \models \forall\,(G_{built} \rightarrow \exists\bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G)_{\mathcal{V}} \mapsto_P \mathcal{N}((H{=}H' \wedge B \wedge C \wedge H' \wedge G)_{\mathcal{V}})$

Fig. 1. Computation Steps of Constraint Handling Rules

that the rule R is *applicable to* H'. A "variant" of a formula is obtained by renaming its variables. "Matching" means that H' is an instance of H, i.e. it is only allowed to instantiate (bind) variables of H but not variables of H'. In the logical notation this is achieved by existentially quantifying only over the fresh variables $\bar{x}$ of the rule to be applied in the condition.
The **Propagate** transition is like the **Simplify** transition, except that it keeps the constraints H' in the state. Trivial nontermination caused by applying the same propagation rule again and again is avoided by applying a propagation rule at most once to the same constraints. A more complex operational semantics that addresses this issue can be found in [Abd97].

Example 1. Let $\leq$ and $<$ be built-in constraint symbols. We define a CHR symbol `max`, where `max(X,Y,Z)` means that `Z` is the maximum of `X` and `Y`:

$$\texttt{max(X,Y,Z)} \Leftrightarrow \texttt{X}\leq\texttt{Y} \mid \texttt{Z}=\texttt{Y}.$$
$$\texttt{max(X,Y,Z)} \Leftrightarrow \texttt{Y}\leq\texttt{X} \mid \texttt{Z}=\texttt{X}.$$
$$\texttt{max(X,Y,Z)} \Rightarrow \texttt{X}\leq\texttt{Z} \wedge \texttt{Y}\leq\texttt{Z}.$$

The first rule states that `max(X,Y,Z)` can be simplified into `Z=Y` in any goal where it holds that $\texttt{X}\leq\texttt{Y}$. Analogously for the second rule. The third rule propagates constraints. It states that `max(X,Y,Z)` unconditionally implies $\texttt{X}\leq\texttt{Z} \wedge \texttt{Y}\leq\texttt{Z}$. Operationally, we add these logical consequences as redundant constraints, the `max` constraint is kept.
To the goal `max(1,2,M)` the first rule is applicable: $\texttt{max(1,2,M)} \mapsto \texttt{M=2}$.
To the goal $\texttt{max(A,B,M)} \wedge \texttt{A<B}$ the first rule is applicable:
$\texttt{max(A,B,M)} \wedge \texttt{A<B} \mapsto \texttt{M=B} \wedge \texttt{A<B}$.
To the goal `max(A,A,M)` both simplification rules are applicable, and in both cases: $\texttt{max(A,A,M)} \mapsto \texttt{M=A}$.
Redundancy from the propagation rule is useful, as the goal `max(A,3,3)` shows:
To this goal only the propagation rule is applicable, and then the first rule: $\texttt{max(A,3,3)} \mapsto \texttt{max(A,3,3)} \wedge \texttt{A}\leq\texttt{3} \mapsto \texttt{A}\leq\texttt{3}$.
Note, that the constraint `3=3` is simplified to true by the built-in constraint solver; according to our assumption about the built-in constraint symbol $=$.

2.4 Confluence

The confluence property of a program guarantees that any computation starting from an arbitrary initial state, i.e. any possible order of rule applications, results in the same final state. Due to space limitations, we can just give an overview on confluence results for CHR programs, for details see [AFM99,Abd97].

Definition 1. A CHR program is called *confluent* if for all states S, S_1, S_2: If $S \mapsto^* S_1$ and $S \mapsto^* S_2$ then S_1 and S_2 are joinable. Two states S_1 and S_2 are called *joinable* if there exist states T_1 and T_2 such that $S_1 \mapsto^* T_1$ and $S_2 \mapsto^* T_2$ and T_1, T_2 are variants of each other.

To analyze confluence of a given CHR program we cannot check joinability starting from any given ancestor state S, because in general there are infinitely many such states. However one can restrict the joinability test to a finite number of "minimal" states based on the following observations: First, adding constraints to a state cannot inhibit the application of a rule as long as the built-in constraints remain consistent (monotonicity property, cf. Lemma 2 in Section 3.2). Hence we can restrict ourselves to ancestor states that consist of the head and guards of two rules. Second, joinability can only be destroyed if one rule inhibits the application of another rule. Only the removal of constraints can affect the applicability of another rule, in case the removed constraint is needed by the other rule. Hence at least 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. This is achieved by equating head atoms in the state.

Definition 2. Given a simplification rule R_1 and an arbitrary (not necessarily different) rule R_2, whose variables have been renamed apart. Let $H_i \wedge A_i$ be the head and C_i be the guard of rule R_i $(i = 1, 2)$. Then a *critical ancestor state of* R_1 *and* R_2 is

$$(H_1 \wedge A_1 \wedge H_2 \wedge (A_1{=}A_2) \wedge C_1 \wedge C_2)_{\mathcal{V}},$$

provided A_1 and A_2 are non-empty conjunctions and $CT \models \exists((A_1{=}A_2) \wedge C_1 \wedge C_2)$.

The application of R_1 and R_2, respectively, to a critical ancestor state of R_1 and R_2 leads to two states that form the so-called *critical pair*.

Definition 3. Let S be a critical ancestor state of R_1 and R_2. If $S \mapsto S_1$ using rule R_1 and $S \mapsto S_2$ using rule R_2 then the tuple (S_1, S_2) is the *critical pair* of R_1 and R_2. A critical pair (S_1, S_2) is *joinable*, if S_1 and S_2 are joinable.

Definition 4. A CHR program is called *terminating*, if there are no infinite computations.

The following theorem from [AFM96,Abd97,AFM99] gives a decidable, sufficient and necessary condition for confluence of a terminating CHR program:

Theorem 1. A terminating CHR program is confluent iff all its critical pairs are joinable.

Example 2. Consider the program for `max` of Example 1. The following critical pair stems from the critical ancestor state[1] $(\mathtt{max(X,Y,Z)} \wedge \mathtt{X} \leq \mathtt{Y})_{[\mathtt{X,Y,Z}]}$ of the first rule and the third one:

$$(S_1, S_2) := (\mathtt{Z{=}Y} \wedge \mathtt{X{\leq}Y} \quad , \quad \mathtt{max(X,Y,Z)} \wedge \mathtt{X{\leq}Y} \wedge \mathtt{X{\leq}Z} \wedge \mathtt{Y{\leq}Z})$$

(S_1, S_2) is joinable since S_1 is a final state and the application of the first rule to S_2 results in S_1.

3 Operational Equivalence

In this section we give sufficient and necessary conditions for equivalence of terminating programs. The following definition states that two programs are operationally equivalent if for each goal, the final state in one program is the same as the final state in the other program.

Definition 5. Let P_1 and P_2 be CHR programs. A state S is P_1, P_2*-joinable*, iff there are two computations $S \mapsto^*_{P_1} S_1$ and $S \mapsto^*_{P_2} S_2$, where S_1 and S_2 are final states, and S_1 and S_2 are variants of each other.
Let P_1 and P_2 be CHR programs. P_1 and P_2 are *operationally equivalent* if all states are P_1, P_2-joinable.

We will see in Section 3.1 that an adaptation of our confluence test - that we call compatibility - does not yield a test for operational equivalence. In Section 3.3, we will be able to give a decidable, sufficient and necessary syntactic condition for operational equivalence of terminating and confluent programs based on results in Section 3.2.
In practice, we want to combine CHR programs which define different CHR symbols, but also have some CHR symbols in common. A typical scenario is that of modules or libraries implementing similar functionality. In the context of CHR, this case arises frequently when constraint solvers written in CHR are combined. Typically, a CHR symbol is only partially defined in a constraint solver.
A closer look at Definition 5 reveals that for these practical scenarios, it is much too strict: States that involve CHR symbols that are defined in one program only, are rarely P_1, P_2-joinable. Therefore, in Section 3.2, we investigate operational equivalence of user-defined constraints that are implemented in different programs.

3.1 Compatibility of Programs

We can use our confluence test to ensure that the different, confluent programs are "compatible": The union of the programs is confluent.

[1] For readability, variables from different rules have been identified to have an overlap.

Definition 6. Let P_1 and P_2 be two confluent and terminating CHR programs and let the union of the two programs, $P_1 \cup P_2$, be terminating. P_1 and P_2 are *compatible* if $P_1 \cup P_2$ is confluent.

Testing the compatibility of P_1 and P_2 means to test the joinability of the critical pairs of $P_1 \cup P_2$, i.e. the critical pairs of P_1 united with the critical pairs of P_2 united with critical pairs coming from one rule in P_1 and one rule in P_2. Note that critical pairs from rules of different programs can only exist, if the heads of the rules have at least one constraint in common.
If the confluence test fails, we can locate the rules responsible for the problem. If the test succeeds, we can just take the union of the rules in the two programs. This means that a common CHR symbol can even be partially defined in the programs which are combined.

Example 3. P_1 contains the following CHR rules defining `max`:

```
max(X,Y,Z) ⇔ X<Y | Z=Y.
max(X,Y,Z) ⇔ X≥Y | Z=X.
```

whereas P_2 has the following definition of `max`:

```
max(X,Y,Z) ⇔ X≤Y | Z=Y.
max(X,Y,Z) ⇔ X>Y | Z=X.
```

We want to know whether the definitions of `max` are compatible. There are three critical ancestor states coming from one rule in P_1 and one rule in P_2:

- `max(X,Y,Z)` $\wedge$ `X<Y` $\wedge$ `X≤Y` stems from the first rule of P_1 and the first rule of P_2.
- `max(X,Y,Z)` $\wedge$ `X≥Y` $\wedge$ `X≤Y` stems from the second rule of P_1 and the first rule of P_2.
- `max(X,Y,Z)` $\wedge$ `X≥Y` $\wedge$ `X>Y` stems from the second rule of P_1 and the second rule of P_2.

Since the critical pairs coming from the critical ancestor states described above are joinable, the two definitions of `max` are compatible. Hence we can just take the union of the rules and define `max` by all four rules.
Note that the compatibility test does not ensure that the constraints are operationally equivalent. In P_1 the goal `max(X,Y,Z)` $\wedge$ `X≥Y` has the following computation:

$$\texttt{max(X,Y,Z)} \wedge \texttt{X}\geq\texttt{Y} \quad \mapsto_{P_1} \quad \texttt{Z=X} \wedge \texttt{X}\geq\texttt{Y}$$

In P_2 the initial state `max(X,Y,Z)` $\wedge$ `X≥Y` is also final state, i.e. no computation step is possible. On the other hand, in P_2 the goal `max(X,Y,Z)` $\wedge$ `X≤Y` has a non-trivial computation, while the goal is a final state in P_1.
The constraint `max` is “operationally stronger” in $P_1 \cup P_2$ than in each program alone, in the sense that more computation steps are possible.

3.2 Equivalence of Constraints

We now introduce a test to ensure that the definitions of the same CHR symbol in different programs are not only compatible, but indeed are operationally equivalent. We first restrict our attention to states that consist of one CHR symbol (only) being common to both programs.

Definition 7. Let c be a CHR symbol. A *c-state* is a state where all user-defined constraints have the same CHR symbol c.

Definition 8. Let c be a CHR symbol defined in two CHR programs P_1 and P_2. P_1 and P_2 are *operationally c-equivalent* if all c-states are P_1, P_2-joinable.

We give now a sufficient syntactic condition for operational c-equivalence of terminating CHR programs. As with confluence, we will try to find a finite subset of states, such that the P_1, P_2-joinability of the subset implies P_1, P_2-joinability of all c-states. As we will see, the similarities with confluence will not go much beyond that, mainly because in operational c-equivalence two different programs are involved.
The following example illustrates, that, first of all, the critical pairs known from confluence (and compatibility) are not the right subset of states to ensure operational equivalence.

Example 4. Let P_1 be the following CHR program:

```
p(a) ⇔ s.
p(b) ⇔ r.
s∧r ⇔ true.
```

and let P_2 consist only of the first two rules.
It is not sufficient for operational equivalence to consider the critical pairs coming from the critical ancestor states `p(a)` and `p(b)`: In P_1 the conjunction `p(a)` $\wedge$ `p(b)` leads to `true`, but in P_2 the goal `s∧r` is a final state.

The example indicates that we not only have to consider c-states, but also those states that can be reached from c-states. Because even if these states can be reached in different programs due to confluence and even if they are final states, there may be contexts (extensions of the states by more constraints) in which the computation can be continued, and it can be continued in different ways in the different programs. The idea is to avoid this by making sure that also the user-defined constraints that occur in these states are operationally equivalent.
For a given CHR symbol c one can safely approximate the set of all CHR symbols that appear in successor states to a c-state by looking at the bodies of rules with c in the head. Based on this idea we introduce the notion of dependency between CHR symbols.

Definition 9. A CHR symbol c *depends directly* on a CHR symbol c', if there is a rule in whose head c appears and in whose body c' appears. A CHR symbol c

depends on a CHR symbol c', if c depends directly on c', or if c depends directly on a CHR symbol d and d depends on c'.
The *dependency set* of a CHR symbol c is the the set of all CHR symbols that c depends on. Let C_{P_1}, C_{P_2} be the dependency sets of c with respect to P_1 and P_2, respectively. Each CHR symbol from $(C_{P_1} \cap C_{P_2}) \cup \{c\}$ is called a *c-dependent CHR symbol*.

Definition 10. Let P_1 and P_2 be CHR programs. The set of *c-critical states* is defined as follows:

$$\{H \wedge C \mid (H \odot C \mid B) \in P_1 \cup P_2, \text{ where } \odot \in \{\Leftrightarrow, \Rightarrow\} \text{ and } H \text{ contains only } c\text{-dependent CHR symbols}\}$$

The set of c-critical states is formed by taking the head and guards of all rules in whose heads c-dependent CHR symbols appear.
In the following we will show that P_1, P_2-joinability of these minimal states is sufficient for P_1, P_2-joinability of arbitrary c-states. Before we can state and prove the theorem, we need several lemmata.
The first lemma states that normalization has no influence on applicability of rules. We therefore can assume in the following that states are normalized except where otherwise noted.

Lemma 1. Let S and S' be states.

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

Proof. Can be found in [AFM99].

The following lemma shows that a computation can be repeated in any context, i.e. with states in which constraints have been added.

Definition 11. The pair of constraints (G_1, G_2) is called *connected via* $\mathcal{V}$ iff all variables that appear both in G_1 and in G_2 also appear in $\mathcal{V}$.

Lemma 2. *[Monotonicity]* If (G, H) is connected via $\mathcal{V}'$ and $G_{\mathcal{V}} \mapsto^* G'_{\mathcal{V}}$, and $\mathcal{V} \subseteq \mathcal{V}'$, then

$$(G \wedge H)_{\mathcal{V}'} \mapsto^* \mathcal{N}((G' \wedge H)_{\mathcal{V}'}).$$

Proof. Can be found in [AFM99].

Next we show that a computation can be repeated in a state where variables have been instantiated according to some equations.

Definition 12. Let C be a conjunction of built-in constraints. Let H and H' be conjunctions of user-defined constraints with disjoint variables. $C[H{=}H']$ is obtained from C by replacing all variables x by the corresponding term t, where $CT \models H{=}H' \rightarrow (x{=}t)$ and x appears in H and t appears in H'.

Lemma 3. Let P be a CHR program and let R be a rule from P with head H and guard C. Let H' be a conjunction of user-defined constraints. Let $(H \wedge H{=}H' \wedge C)_{\mathcal{V}}$ and $(H' \wedge C[H{=}H'])_{\mathcal{V}'}$ be intial states, where H and H' have disjoint variables. If $CT \models \exists \bar{x}(H = H' \wedge C)$, where $\bar{x}$ are the variables appearing in H, and $(H \wedge H = H' \wedge C)_{\mathcal{V}} \mapsto_P^* G_{\mathcal{V}}$, then $(H' \wedge C[H{=}H'])_{\mathcal{V}'} \mapsto_P^* G_{\mathcal{V}'}$.

Proof. The claim holds due to the equality propagation property of the normalization function $\mathcal{N}$ and according to Lemma 1. A detailed proof can be found in [AF99].

Next we show that a computation can be repeated in a state where redundant built-in constraints have been removed.

Lemma 4. Let C be a conjunction of built-in constraints. If $H \wedge C \wedge G \mapsto^* S$ and $CT \models \forall\, (G_{built} \rightarrow C)$ then $H \wedge G \mapsto^* S$.

Proof. This is a consequence of the following claim: If $H \wedge C \wedge G \mapsto S$ and $CT \models \forall\, (G_{built} \rightarrow C)$ then $H \wedge G \mapsto S$. This claim can be proven by analyzing each kind of computation step [AF99].

Finally, the last Lemma refers to joinability of c-critical states.

Definition 13. Let $C = \bigwedge\limits_{i=1}^{n} C_i$ be a conjunction of constraints, π a permutation on $[1, \ldots, n]$, where $0 \leq m \leq n$, then $\bigwedge\limits_{i=1}^{m} C_{\pi_i}$ is a *subconjunction* of C.

Lemma 5. Let P_1 and P_2 be terminating CHR programs defining a CHR symbol c and let G be a goal. If all c-critical states are P_1, P_2-joinable and there is a rule in P_1 that is applicable to G_{user} then there is a rule in P_2 that is applicable to a subconjunction of G_{user}.

Proof. Can be found in [AF99].

We are now ready to state and prove the main theorem of the paper, that gives a sufficient condition for operational c-equivalence.

Theorem 2. Let c be a CHR symbol defined in two confluent and terminating CHR programs P_1 and P_2. Then the following holds: P_1 and P_2 are operationally c-equivalent if all c-critical states are P_1, P_2-joinable.

The proof can be found in [AF99].

We now give an example of two operationally equivalent user-defined constraints.

Example 5. The constraint `sum(List,Sum)` holds if `Sum` is the sum of elements of a given list `List`. The CHR symbol `sum` can be implemented in different ways. Let P_1 be the following CHR program:

```
sum([],Sum) ⇔ Sum=0.
sum([X|Xs],Sum) ⇔  sum(Xs,Sum1) ∧ Sum = Sum1 + X.
```

Let P_2 be a CHR program defining `sum` using an auxiliary CHR symbol `sum1`:

```
sum([],Sum) ⇔ Sum = 0.
sum([X|Xs],Sum) ⇔  sum1(X,Xs,Sum).
sum1(X,[],Sum) ⇔ Sum = X.
sum1(X,Xs,Sum) ⇔ sum(Xs,Sum1) ∧ Sum = Sum1 + X.
```

`sum([],Sum)` and `sum([X|Xs],Sum)` are the `sum`-critical states coming from P_1 and P_2. The `sum`-critical states are P_1, P_2-joinable:
For the `sum`-critical state `sum([],Sum)` the final state is `Sum = 0` in both P_1 and P_2.
A computation of the `sum`-critical state `sum([X|Xs],Sum)` in P_1 proceeds as follows:
`sum([X|Xs],Sum)` $\mapsto_{P_1}$ $\mathtt{sum(Xs,Sum1)} \wedge \mathtt{Sum} = \mathtt{Sum1} + \mathtt{X}$
A computation of the same initial state in P_2 results in the same final state:
`sum([X|Xs],Sum)` $\mapsto_{P_2}$ $\mathtt{sum1(X,Xs,Sum)} \mapsto_{P_2} \mathtt{sum(Xs,Sum1)} \wedge \mathtt{Sum} = \mathtt{Sum1} + \mathtt{X}$
Since all `sum`-critical states are P_1, P_2-joinable, P_1 and P_2 are operationally `sum`-equivalent.

The next example shows why our joinability test for critical states is a sufficient, but not necessary condition for operational equivalence.

Example 6. Let P_1 be the following CHR program

```
p(X) ⇔  X>0 | q(X).
q(X) ⇔  X<0 | true.
```

and let P_2 be the following one

```
p(X) ⇔  X>0 | q(X).
q(X) ⇔  X<0 | false.
```

P_1 and P_2 are operationally `p`-equivalent, but the `p`-critical state $\mathtt{q(X)} \wedge \mathtt{X} < \mathtt{0}$ is not P_1, P_2-joinable.

The reason that we can only give a sufficient, but not necessary condition for operational c-equivalence in the general class of CHR programs is that the dependency relation between user-defined constraints only approximates the actual set of user-defined constraints that occur in states that can be reached from a c-state.

A sufficient and necessary condition: In practice, one is often interested to compare constraint solvers which have only one CHR symbol in common. In this case we can give a decidable, sufficient and necessary condition.

Theorem 3. Let c be the only CHR symbol defined in two confluent and terminating CHR programs P_1 and P_2. P_1 and P_2. Then the following holds: P_1 and P_2 are operationally c-equivalent iff all c-critical states are P_1, P_2-joinable.

Proof. "$\Longrightarrow$" direction: Let P_1 and P_2 be operationally c-equivalent. We prove by contradiction that all c-critical states are P_1, P_2-joinable: Assume that $H \wedge C$ is a c-critical state that is not P_1, P_2-joinable, where H is the head of a rule from $P_1 \cup P_2$ and C its guard.
Since P_1 and P_2 have only c in common, the constraint symbol c is the only c-dependent CHR symbol, i.e. $(C_{P_1} \cap C_{P_2}) \cup \{c\} = \{c\}$. Therefore $H \wedge C$ is a c-state. This contradicts the prerequisite that P_1 and P_2 are operationally c-equivalent.
"$\Longleftarrow$" direction: This is a special case of Theorem 2.

Theorem 3 gives a decidable characterization of the c-equivalent subset of terminating and confluent CHR programs: P_1, P_2-joinability of a given c-critical state is decidable for a terminating CHR program and there are only finitely many c-critical states.

Example 7. The user-defined constraint `range(X,Min,Max)` holds if `X` is between `Min` and `Max`.
Let P_1 be a CHR program defining `range` using the CHR symbol `max`:

```
max(X,Y,Z) ⇔ X<Y | Z=Y.
max(X,Y,Z) ⇔ X≥Y | Z=X.

range(X,Min,Max) ⇔ max(X,Min,X) ∧ max(X,Max,Max).
```

Let P_2 be a program defining `range` using the built-in constraint symbols $<$, $\leq$:

```
range(X,Min,Max) ⇔ Max<Min | false.
range(X,Min,Max) ⇔ Min≤Max | Min≤X ∧ X≤Max.
```

P_1 and P_2 are not operationally `range`-equivalent, since the `range`-critical state `range(X, Min, Max)` coming from P_1 is not P_1, P_2-joinable: `range(X, Min, Max)` can be reduced to `max(X,Min,X) ∧ max(X,Max,Max)` in P_1. In P_2 the answer for the state `range(X, Min, Max)` is the state itself, because no rule is applicable.
P_1 is "operationally stronger" than P_2, since the computation step in P_1 does not require that the values of `Max` and `Min` are known. This can be exemplified by the goal `range(5,6,Max)`. The inconsistency of the goal can be detected in P_1. In P_2, `range(5,6,Max)` is a final state.

3.3 Equivalence of Programs

Based on the condition presented above for the operational equivalence of constraints we can also give a decidable, sufficient and necessary condition for operational equivalence of terminating and confluent programs.
However, it is not enough to consider the union of all c-critical states for all common CHR symbols c, as the following example illustrates.

Example 8. Let P_1 be

```
p ⇔ s.
s∧q ⇔ true.
```

and let P_2 be

```
p ⇔ s.
s∧q ⇔ false.
```

P_1 and P_2 have three common CHR symbols, `p`, `s` and `q`. `s` and `p` are the `p`-dependent constraint symbols. There are no `s`-dependent CHR symbols except `s` itself. Analogously for `q`.
`p` is the only `p`-critical state. It is P_1, P_2-joinable. There is no `s`-critical state, since `q` is not a `s`-dependent CHR symbol. Analogously for `q`.
Hence all `p`-, `s` and `q`-critical states are P_1, P_2-joinable, but the programs are not operationally equivalent. `s∧q` leads in P_1 to `true` and with P_2 to `false`.

Still we can prove the operational equivalence of two programs by adapting the definition of c-critical states:

Definition 14. Let P_1 and P_2 be CHR programs. The *set of critical states of* P_1 *and* P_2 is defined as follows:

$$\{H \wedge C \mid (H \odot C \mid B) \in P_1 \cup P_2, \text{ where } \odot \in \{\Leftrightarrow, \Rightarrow\}\}$$

Theorem 4. Let P_1 and P_2 be terminating and confluent programs. P_1 and P_2 are operationally equivalent iff all critical states of P_1 and P_2 are P_1, P_2-joinable.

Proof. Follows the proof of Theorem 3.

Relationships. Operational equivalence of two confluent and terminating CHR programs implies their compatibility, since operational equivalence of P_1 and P_2 implies the confluence of $P_1 \cup P_2$. The converse does not hold, as the programs of Example 3 show. Furthermore operational equivalence of two CHR programs implies the operational c-equivalence of all common constraints, since the set of critical states is a superset of the union of all sets of the c-critical states. The converse does not hold, as the programs of Example 8 show.

4 Conclusion

We introduced the notion of operational equivalence of CHR programs. We gave a decidable, sufficient and necessary syntactic condition for operational equivalence of terminating and confluent CHR programs. A decidable, sufficient and necessary condition for confluence of a terminating CHR programs was given in earlier work [AFM96,Abd97,AFM99]. We have also shown that an extension of the confluence notion to two programs, called compatibility, is not sufficient.
For practical reasons, we also investigated a notion of operational equivalence for user-defined constraints that are defined in different programs. We gave a sufficient syntactic condition for constraints defined in terminating and confluent CHR programs. For programs which have only one user-defined constraint in common, we were able to give a sufficient and necessary syntactic condition.

Future work aims to enlarge the class of CHR programs for which we can give a sufficient and necessary syntactic condition for operational equivalence. We also plan to investigate the relationship between operational equivalence and logical equivalence of CHR programs. The complication is that different programs have different signatures and are therefore hard to compare logically. Roughly, operational equivalence seems to imply logical equivalence (but not the other way round, see e.g. Example 3). Furthermore, operational equivalence together with completion [AF98] provide a good starting point for investigating partial evaluation, and program transformation in general, of constraint solvers.

Acknowledgements. We would like to thank Norbert Eisinger and Holger Meuss for useful comments on a preliminary version of this paper.

References

[Abd97] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Third International Conference on Principles and Practice of Constraint Programming, CP97*, LNCS 1330. Springer-Verlag, November 1997.

[ACM96] ACM. The constraint programming working group. Technical report, ACM-MIT SDRC Workshop, Report Outline, 1996.

[AF98] S. Abdennadher and T. Frühwirth. On completion of constraint handling rules. In *4th International Conference on Principles and Practice of Constraint Programming, CP98*, LNCS 1520. Springer-Verlag, 1998.

[AF99] S. Abdennadher and T. Frühwirth. Operational equivalence of constraint handling rules. Research report PMS-FB-1999-4, Computer Science Department, University of Munich, 1999.

[AFM96] S. Abdennadher, T. Frühwirth, and H. Meuss. On confluence of constraint handling rules. In *2nd International Conference on Principles and Practice of Constraint Programming, CP96*, LNCS 1118. Springer-Verlag, August 1996.

[AFM99] S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal, Special Issue on the Second International Conference on Principles and Practice of Constraint Programming*, 4(2), May 1999.

[EGM98] S. Etalle, M. Gabrielli, and M. Meo. Unfold/fold transformations of CCP programs. In *9th International Conference on Concurrency Theory*, 1998. Corrected version.

[Frü98] T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, pages 95–138, October 1998.

[GLM95] M. Gabbrielli, G. Levi, and M. Chiara Meo. Observable behaviors and equivalences of logic programs. *Information and Computation*, 122(1):1–29, October 1995.

[Mah86] M. J. Maher. Equivalences of logic programs. In *Proceedings of Third International Conference on Logic Programming*, Berlin, 1986. Springer.