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

Ludwig-Maximilians-Universität München

Confluence and Semantics of Constraint Simplification Rules

Slim Abdennadher, Thom Frühwirth, Holger Meuss

To appear in Constraints Journal 1998
http://www.pms.informatik.uni-muenchen.de/publikationen
Forschungsbericht/Research Report PMS-FB-1997-20, Mai 1997

Confluence and Semantics of Constraint Simplification Rules

Slim Abdennadher, Thom Frühwirth and Holger Meuss
Computer Science Department, Ludwig-Maximilians-University
Oettingenstrasse 67, D-80538 Munich, Germany
{*Slim.Abdennadher, Thom.Fruehwirth*} *@informatik.uni-muenchen.de*
meuss@cis.uni-muenchen.de
http://www.pst.informatik.uni-muenchen.de/personen/fruehwir/cwg.html

(Received ..... ; Accepted in final form .....)

Abstract.
Constraint Simplification Rules (CSR) is a subset of the Constraint Handling Rules (CHR) language. CHR is a powerful special-purpose declarative programming language for writing constraint solvers. The CSR subset of CHR forms essentially a committed-choice language consisting of guarded rules with multiple heads that replace constraints by simpler ones until they are solved. This paper gives declarative and operational semantics as well as soundness and completeness results for CSR programs.
In this paper, we introduce a notion of confluence for CSR programs. Confluence is an essential syntactical property of any constraint solver. It ensures that the solver will always compute the same result for a given set of constraints independent of which rules are applied. It also means that it does not matter for the result in which order the constraints arrive at the constraint solver.
We give a decidable, sufficient and necessary syntactic condition for confluence of terminating CSR programs. Moreover, as shown in this paper, confluence of a program implies consistency of its logical meaning (under a mild restriction).
Keywords: constraint reasoning, semantics of programming languages, committed-choice languages, confluence, determinism, program analysis.

1. Introduction

Constraint-based programming languages, be it constraint logic programming (CLP) [JL87, Mah87, vH91, FHK$^+$92, JM94] or committed-choice concurrent constraint logic (C^{4}L) programming[1] [Mah87, Sha89, SRP91, Sar93, JM94], enjoy both elegant theoretical properties and practical success. 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 the constraints, and detect their inconsistency. Intuitively, constraints represent elementary relationships between variables and values, for example equality or some order relationships. Clearly, the abilities and quality of the constraint solver play an essential role in constraint-based programming.

[1] There is no consistent terminology in the literature for this class of programming languages: You may drop "logic" and either "committed-choice" or "concurrent".

In the beginning, constraint solving was "hard-wired" in a built-in constraint solver written in a low-level language, termed the "black-box" approach. While efficient, this approach makes it hard to modify a solver or build a solver over a new domain, let alone reason about and analyze it. As the behavior of the solver can neither be inspected by the user nor explained by the computer, debugging of constraint-based programs is hard. Also, one lesson learned from practical applications is that constraints are often heterogeneous and application specific.
Several proposals have been made to allow more flexibility and customization of constraint solvers, often termed "glass-box" approaches [CD93, vH91]. The most far-reaching proposal is the "no-box" approach: Constraint Handling Rules (CHR) [Frü95] is a high-level language for writing constraint solvers either from scratch or by modifying existing solvers. The CSR (Constraint Simplification Rules) subset of CHR is essentially a C^4L language consisting of guarded rules with multiple heads that replace (conjunctions of) constraints by simpler ones until they are solved. With single-headed CSR rules alone, unsatisfiability of constraints could not always be detected (e.g. `X<Y,Y<X`).

In contrast to typical general-purpose C^4L languages, CSR 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. We give soundness and completeness results for a class of CSR programs.
There are C^4L languages that share their semantics with CSR. The *Guarded Rules* [Smo91] correspond to single headed CSR. However, they are only used as "shortcuts" (lemmata) for predicates, not as definitions for user-written constraints. Interestingly, in [Smo91] the built-in constraint system is defined as a terminating and determinate reduction system. Hence it could be implemented by CSR.
Also [AKP94] relies on a kind of guarded rules, emphasizing their use as a programming language on its own. [AKP94] shows that guarded rule programs can be given a logical meaning that is a consistent theory, provided that the guards satisfy a logical condition called compatibility and a kind of closed-world assumption. Since CSR allows multiple heads, it cannot have such a closed-world assumption.

Typically, more than one CSR 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 CSR rules is applied. This essential property of any constraint solver will be called confluence. Without confluence, one computation may detect inconsistency while another might just simplify the same constraints into a more complex constraint. Confluence also implies that it does not matter in which order the constraints arrive at the constraint solver.

Consider the following rules from a constraint solver for interval domains as an example. The intervals are associated with variables, written `X::A..B`, which means `A`$\leq$`X` $\wedge$ `X`$\leq$`B`. The first rule ensures that the interval for `X` is non-empty, the second rule intersects two intervals for the same variable:

```
X::A..B <=> A>B | false.
X::A..B, X::C..D <=> true | maximum(A,C,E), minimum(B,D,F),
                            X::E..F.
```

The first CSR rule reads: If the guard `A>B` holds then replace the constraint `X::A..B` by the constraint `false` exhibiting its inconsistency. The program consisting of these two rules is confluent. Adding the seemingly harmless rule that handles a variable whose value is uniquely determined by its interval,

```
X::A..A <=> true | X≐A.
```

results in a program that is not confluent anymore: The constraints `X::3..3, X::5..7` can be simplified to `X::5..3` by the second rule. This constraint in turn simplifies to `false` with the first rule, so that the inconsistency of the initial constraints is exhibited. On the other hand, applying the newly added rule to the first constraint leads to `X≐3, X::5..7`. No more simplification is possible, the inconsistency is left implicit.

We will introduce a decidable, sufficient and necessary syntactic condition for confluence of terminating CSR programs. This condition adopts the notion of critical pairs as known from term rewriting systems [DOS88, KK91, Pla93]. A straightforward translation of the results in this field was not possible, because the CSR formalism gives rise to phenomena not appearing in this combination in research on confluence in term rewriting systems. These include the way in which variables can occur in a rule and the existence of global knowledge. CSR programs are more powerful than the classical conditional rewriting, because they use an additional context which is the built-in constraint store.

A practical application of our definition of confluence lies in program analysis, where we can identify non-confluent parts of CSR programs by examining the so-called critical pairs between rules. Programs with non-confluent parts are likely to represent an ill-defined constraint solver. That a decidable confluence test exists is a clear advantage of CSR over black-box approaches. Since our test for confluence is decidable for terminating programs, it can also be used to identify the parts of arbitrary terminating C^4L programs that have a declarative semantics in our sense.

On the theoretical side we also show that confluence implies consistency of the logical meaning of a CSR program (under a mild restriction). Furthermore we can improve on completeness, if a CSR program is confluent (and terminating).

Our approach is orthogonal to the work in program analysis for C^{4}L languages as in [MO95, CFMW97, FGMP95], where a different, less rigid notion of confluence is defined: A committed-choice program is confluent, if different process schedulings give rise to the same set of possible outcomes. The idea of [MO95, CFMW97] is to introduce a non-standard semantics, which is confluent for all committed-choice programs.

This paper is organized as follows: The next section introduces the syntax of Constraint Simplification Rules (CSR), their declarative and operational semantics. Then we relate the declarative and operational semantics of CSR programs by giving soundness and completeness results. Section 3 presents our notion of confluence for CSR. In section 4 we show that confluence implies consistency of the logical meaning of a program. In section 5 we show how confluence leads to a strong completeness result for finite failure. Finally, we conclude with a summary and directions for future work. The appendix contains the main proofs, which are quite long. A preliminary short version of this paper was presented at CP'96 [AFM96].

2. Syntax and Semantics

In this section we give syntax and semantics as well as soundness and completeness results for Constraint Simplification Rules (CSR). We assume some familiarity with C^{4}L programming [JL87, JM94, SRP91, Sar93, Sha89]. Constraints are considered to be special first-order predicates. We will distinguish between two classes of constraints. *Built-in* constraints are those handled by an already existing, predefined constraint solver. *User-defined* constraints are those defined by a CSR program.

Definition 2.1. A *CSR program* is a finite set of constraint simplification rules. A *(constraint) simplification rule* is of the form

$$H_1, \ldots, H_i \Leftrightarrow G_1, \ldots, G_j \mid B_1, \ldots, B_k \quad (i > 0, j \geq 0, k \geq 0),$$

where the head $H_1, \ldots, H_i$ is a non-empty conjunction[2] of user-defined constraints, the guard[3] $G_1, \ldots, G_j$ 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 built-in and user-defined constraints are called *goals*.

Without loss of generality we assume the rules of the CSR program in question to have disjoint sets of variables. In examples we may disregard this agreement for ease of reading.

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

[3] The commit symbol "|" should not be confused as standing for disjunction as in grammar formalisms and some Prolog dialects.

2.1. Declarative Semantics

In general, C^4L programs do not have a declarative semantics [AKP94]. Typically, Clark's completion is used to describe the logical meaning of a program. For CSR, we chose a different declarative semantics, since Clark's completion cannot be used for CSR's multiple heads. This semantics has also been proposed for guarded rules [AKP94, Smo91].
The declarative semantics of a CSR program P is given by a conjunction of universally quantified logical formulae (one for each rule), $\mathcal{P}$, and a consistent built-in theory CT which determines the meaning of the built-in constraints appearing in the program. The constraint theory CT is expected to include a constraint $\doteq$ for syntactic equality (e.g. by Clark's equality theory CET [Cla78]) and the constraints *true* and *false*.

Definition 2.2. The logical meaning of a simplification rule is a logical equivalence provided the guard holds

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

where $\bar{x}$ is the sequence of variables occuring in $H_1, \ldots, H_n$ and $\bar{y}$ are the other variables occuring in $G_1, \ldots, G_j$ and $\bar{z}$ are the variables occuring in $B_1, \ldots, B_k$ only.

Example 2.1. Now let us extend a given constraint solver for the constraints $\leq$ and $\doteq$ with a constraint `maximum(X,Y,Z)` which holds, if `Z` is the maximum of `X` and `Y`. The following rules could be part of the CSR program:

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

The first rule states that `maximum(X,Y,Z)` can be replaced by `Z≐Y` provided it holds that `X≤Y`.
Now assume there is a typo in the body of the second rule:

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

The logical meaning of this CSR program is the theory

```
∀ X,Y,Z (X ≤ Y → (maximum(X,Y,Z) ↔ Z ≐ Y))
∀ X,Y,Z (Y ≤ X → (maximum(X,Y,Z) ↔ Y ≐ X))
```

together with an appropriate constraint theory describing $\leq$ as an order relation. The logical meaning $\mathcal{P}$ of this program is not a consistent theory. This can be exemplified by the atomic formula $\mathrm{maximum}(1,1,0)$, which is logically equivalent to $0 \doteq 1$ (and therefore *false*) using the first formula. Using the second formula, however $\mathrm{maximum}(1,1,0)$ is logically equivalent to $1 \doteq 1$ (and therefore true).

2.2. Operational Semantics

The operational semantics of CSR is a straightforward extension of the usual one for C[4]L languages [JM94] to multiple head atoms. We define the operational semantics of a given CSR program P by a transition system that models the operations of the constraint solver defined by P. To keep the semantics simple, we require from now on that those guard constraints containing variables which do appear in the body but not in the head have to appear in the body again. This is no real restriction, since a general rule can be translated into a restricted rule by simply repeating the guard constraints in the body.

Example 2.2. A CSR rule of the form `p(X)` $\Leftrightarrow$ `Y = 3 | X = Y` must be translated to `p(X)` $\Leftrightarrow$ `Y = 3 | X = Y, Y = 3`.

2.2.1. *States*

Definition 2.3. A *state* is a tuple

$$\langle Gs, C_U, C_B, \mathcal{V}\rangle.$$

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) stores*, respectively. $\mathcal{V}$ is a sequence of variables. An empty goal 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.

Definition 2.4. A variable X appearing in a state $\langle Gs, C_U, C_B, \mathcal{V}\rangle$ 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 2.5. The *logical meaning* of a state $\langle Gs, C_U, C_B, \mathcal{V}\rangle$ 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.

2.2.2. *A Normalized Form for States*

We will assume that states are in a normalized form that abstracts away the specifics of the built-in constraint solver: The normalized form considers those states equivalent that impose the same built-in constraints on the goal and on the user-defined constraints store. We model the normalization with a function that maps equivalent states into a syntactically unique representative state. The normalization function normalizes the built-in constraint store, projects out strictly local variables, and propagates implied equations all over the state. Most built-in constraint solvers naturally support this functionality since they work with normalized forms anyway. For the following theorems and proofs it is important to make the requirements on the normalization function more precise.

Definition 2.6. A function $\mathcal{N} : \mathcal{S} \rightarrow \mathcal{S}$, where $\mathcal{S}$ is the set of all states, is a *normalization function*, if it fulfills the following conditions. Let $\mathcal{N}(\langle Gs, C_U, C_B, \mathcal{V}\rangle) = \langle Gs', C'_U, C'_B, \mathcal{V}\rangle$. We assume that there is a fixed order on variables appearing in a state such that global variables are ordered as in $\mathcal{V}$ and precede all local variables.

- *Equality propagation:* Gs' and C'_U derive from Gs and C_U by replacing all variables X, that are uniquely determined in C_B [JM94], i.e. for which $CT \models \forall\ (C_B \rightarrow X \doteq t)$[4] holds, by the corresponding term t, except if t is a variable that comes after X in the variable order.

- *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 $\langle Gs', C'_U, C_B, \mathcal{V}\rangle$.

- *Uniqueness:* If

$$\begin{aligned} \mathcal{N}(\langle Gs_1, C_{U1}, C_{B1}, \mathcal{V}\rangle) &= \langle Gs'_1, C'_{U1}, C'_{B1}, \mathcal{V}\rangle \text{ and} \\ \mathcal{N}(\langle Gs_2, C_{U2}, C_{B2}, \mathcal{V}\rangle) &= \langle Gs'_2, C'_{U2}, C'_{B2}, \mathcal{V}\rangle \text{ and} \\ CT \models (\exists \bar{x} C_{B1}) &\leftrightarrow (\exists \bar{y} C_{B2}), \end{aligned}$$

  holds, where $\bar{x}$ and $\bar{y}$, respectively, are the strictly local variables of the two states, then:

$$C'_{B1} = C'_{B2}.$$

The syntactical form of the result of normalization does not matter, as long as the three conditions, above all uniqueness, hold. An important property of $\mathcal{N}$ is that it preserves the logical meaning of states.

[4] $\forall F$ is the universal closure of a formula F, likewise is $\exists F$ the existential closure of F.

Lemma 2.1. Let be

$$\mathcal{N}(\langle Gs, C_U, C_B, \mathcal{V}\rangle) = \langle Gs', C_U', C_B', \mathcal{V}\rangle.$$

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.

Proof. The claim follows from the following three assertions:

$$\bar{x}' \subseteq \bar{x} \tag{1}$$

$$CT \models \forall\ (C_B \rightarrow ((Gs \wedge C_U) \leftrightarrow (Gs' \wedge C_U'))) \tag{2}$$

$$CT \models \forall\ (\exists\bar{y}C_B \leftrightarrow C_B') \text{ and } \bar{y} \subseteq \bar{x}, \tag{3}$$

where $\bar{y}$ are the strictly local variables in $\langle Gs', C_U', C_B, \mathcal{V}\rangle$. Assertion (1) holds because the normalization function $\mathcal{N}$ does not introduce new variables due to the projection property. (2) holds, because CT contains equality and $Gs' \wedge C_U'$ derive from $Gs \wedge C_U$ by substitutions prescribed by C_B. (3) follows from the uniqueness property of $\mathcal{N}$. ($\bar{y}$ are the strictly local variables in $\langle Gs', C_U', C_B, \mathcal{V}\rangle$.) The claim then directly follows from the assertions (1), (2) and (3). □

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* and likewise a valid built-in store by *true*.
A property of $\mathcal{N}$ is that it will eliminate all strictly local variables:

Example 2.3. Let

$$\mathcal{N}(\langle p(Z), \top, X \doteq Z, [X]\rangle) = \langle p(X), \top, C_B, [X]\rangle.$$

Because $CT \models \forall\ (\exists Z(X \doteq Z) \leftrightarrow \text{true})$, the uniqueness condition implies the following:

$$\mathcal{N}(\langle p(X), \top, \text{true}, [X]\rangle) = \langle p(X), \top, C_B, [X]\rangle,$$

Therefore we know that C_B must be *true*, because $\mathcal{N}$ cannot introduce new variables.

Definition 2.7. The pair (C_1, C_2) (C_1 and C_2 are conjunctions of constraints) is called *connected* in the sequence $\mathcal{V}$ iff all variables that appear in C_1 and C_2 also appear in $\mathcal{V}$.

The following lemma shows that $\mathcal{N}$ is to a certain degree compatible with adding constraints to the built-in store:

Lemma 2.2. If C is a conjunction of built-in constraints and (C, C_B) is connected in $\mathcal{V}$ and $\mathcal{N}(\langle Gs, C_U, C_B, \mathcal{V}\rangle) = \langle Gs', C'_U, C'_B, \mathcal{V}\rangle$ then

$$\mathcal{N}(\langle Gs, C_U, C_B \wedge C, \mathcal{V}\rangle) = \mathcal{N}(\langle Gs', C'_U, C'_B \wedge C, \mathcal{V}\rangle).$$

This claim is proven by analyzing the strictly local variables of the states. The connectedness requirement in the lemma above reflects the sensitivity of $\mathcal{N}$ to strictly local variables. It guarantees that equality constraints involving variables appearing in the added constraint C are not removed by $\mathcal{N}$ due to locality.

2.2.3. *Computation Steps*

The aim of the computation is to incrementally reduce arbitrary states to states that contain no more goals in the goal store and a maximally simplified user-defined constraint store (with regard to a given program P). Given a CSR program P we define the transition relation $\mapsto_P$[5] by introducing three kinds of computation steps (Figure 1).

Transitions

Solve

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

Introduce

$$\frac{C \text{ is a user-defined constraint}}{\langle C \wedge Gs, C_U, C_B, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle Gs, C \wedge C_U, C_B, \mathcal{V}\rangle)}$$

Simplify

$(H \Leftrightarrow G \mid B)$ is a fresh variant of a rule in P with the variables $\bar{x}$

$$\frac{CT \models \forall\, (C_B \rightarrow \exists \bar{x}(H \dot{=} H' \wedge G))}{\langle Gs, H' \wedge C_U, C_B, \mathcal{V}\rangle \mapsto \mathcal{N}(\langle Gs \wedge B, C_U, H \dot{=} H' \wedge C_B, \mathcal{V}\rangle)}$$

Figure 1. Computation Steps

Notation: Capital letters denote conjunctions of constraints. By equating two constraints $(c(t_1, \ldots, t_n) \dot{=} c(s_1, \ldots, s_n))$, we mean $t_1 \dot{=} s_1 \wedge \ldots \wedge t_n \dot{=} s_n$. By

[5] In the rest of the paper, we will drop P for simplicity.

$(p_1 \wedge \ldots \wedge p_n) \doteq (q_1 \wedge \ldots \wedge q_n)$ 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*.
In the **Solve** transition, the built-in solver simplifies the built-in store after adding a new constraint C that was found in the goal store. **Introduce** transports a user-defined constraint C from the goal store into the user-defined constraint store. To **Simplify** user-defined constraints H' means to replace them by the body B of a fresh variant[6] of a simplification rule $(H \Leftrightarrow G \mid B)$ from the program, provided H' matches[7] the head H and the resulting guard G is implied by the built-in constraint store, and finally to normalize the resulting state.

Definition 2.8. An *initial state* for a goal G is of the form:

$$\langle G, \top, \text{true}, \mathcal{V} \rangle,$$

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

$$\langle G, C_U, \text{false}, \mathcal{V} \rangle,$$

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

$$\langle \top, C_U, C_B, \mathcal{V} \rangle$$

with no computation step possible anymore and C_B not *false* (such a state 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 initial state for G and ending in a final state or diverging. A computation is *finitely failed*, if it is finite and its final state is failed.

Example 2.4. Remember the correct rules for `maximum`:

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

A computation of the goal `maximum(1,1,Z)` proceeds as follows (using the first rule):

[6] Two expressions are variants, if they can be obtained from each other by a variable renaming. A fresh variant contains only new variables.
[7] Matching rather than unification is the effect of the existential quantification over the head equalities.

$$
\begin{array}{ll}
 & \langle \mathtt{maximum(1,1,M)}, \top, \mathtt{true}, [\mathtt{M}] \rangle \\
\mapsto (\textbf{Introduce}) & \mathcal{N}(\langle \top, \mathtt{maximum(1,1,M)}, \mathtt{true}, [\mathtt{M}] \rangle) = \\
 & \langle \top, \mathtt{maximum(1,1,M)}, \mathtt{true}, [\mathtt{M}] \rangle \\
\mapsto (\textbf{Simplify}) & \mathcal{N}(\langle \mathtt{Z} \doteq \mathtt{Y}, \top, \mathtt{X} \doteq \mathtt{1} \wedge \mathtt{Y} \doteq \mathtt{1} \wedge \mathtt{Z} \doteq \mathtt{M}, [\mathtt{M}] \rangle) = \\
 & \langle \mathtt{M} \doteq \mathtt{1}, \top, \mathrm{true}, [\mathtt{M}] \rangle = \\
\mapsto (\textbf{Solve}) & \mathcal{N}(\langle \top, \top, \mathtt{M} \doteq \mathtt{1}, [\mathtt{M}] \rangle) = \\
 & \langle \top, \top, \mathtt{M} \doteq \mathtt{1}, [\mathtt{M}] \rangle
\end{array}
$$

Lemma 2.3. 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

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

2.3. Soundness and Completeness

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

Definition 2.11. A *computable constraint* of G is the logical meaning of a state which appears in a computation of G. The logical meaning of a final state is called *answer constraint.*

The results in this section are relatively straightforward because a computation step produces only logically equivalent states.
The following lemma and theorem are direct consequences of Lemma A.1 (to be found in the appendix).

Lemma 2.4. Let P be a CSR program and G be a goal. Then for all computable constraints C_1 and C_2 of G the following holds:

$$P, CT \models C_1 \leftrightarrow C_2.$$

Theorem 2.1 (Soundness). Let P be a CSR program and G be a goal. If G has a computation with answer constraint C then

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

The following theorem is stronger than the completeness result for constraint logic programming languages (CLP) as presented in [Mah87]. We can reduce the disjunction in the strong completeness theorem presented there to a single disjunct in our theorem. This is possible, since our declarative semantics is stronger and consequently, according to Lemma 2.4, all computable constraints of a given goal are equivalent (Figure 2).

(A)
$$\begin{tikzcd} & S \arrow[dl, "*"'] \arrow[dr, "*"] & \\ S_1 & \not\leftrightarrow & S_2 \end{tikzcd}$$

(B)
$$\begin{tikzcd} & S \arrow[dl, "*"'] \arrow[dr, "*"] & \\ S_1 & \leftrightarrow & S_2 \end{tikzcd}$$

Figure 2. Logical Relationship of Computable Answers in CLP (A) and CSR (B)

Theorem 2.2 (Completeness). Let P be a CSR program and G be a goal with at least one finite computation. If $\mathcal{P}, CT \models \forall\ (C \leftrightarrow G)$, then G has a computation with answer constraint C' such that

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

Proof. G has at least one finite computation. Let C' be the answer constraint of G resulting from this computation.
By the soundness Theorem 2.1 the following holds:

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

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

The completeness theorem does not hold, if G has no finite computations.

Example 2.5. Let P be the following CSR program:

```
p ⇔ p.
```

Let G be `p`. It holds that $\mathcal{P}, CT \models \mathtt{p} \leftrightarrow \mathtt{p}$. However, G has only one infinite computation.

Corollary 2.1. Let P be a CSR program and G be a goal.
If G has a finitely failed computation, then $\mathcal{P}, CT \models \neg\exists G$.
Proof. If G has a finitely failed computation, then G has a computation with answer constraint equivalent to *false*. By Theorem 2.1 we have that $\mathcal{P}, CT \models \forall\,(\text{false} \leftrightarrow G)$, hence $\mathcal{P}, CT \models \forall\neg G$, which is equivalent to $\mathcal{P}, CT \models \neg\exists G$. □

The converse of Corollary 2.1 does not hold in general:

Example 2.6. Let P be the following CSR program

```
p ⇔ q.
p ⇔ false.
```

$\mathcal{P}, CT \models \neg q$, but q has no finitely failed computation. We will see that confluence will improve on this situation.

3. Confluence

We have already shown in the previous section that in every CSR program, the result of a computation of a given goal will always have the same meaning. However it is not guaranteed that the result is syntactically the same. In particular, a solver may be complete with one order of rule applications but incomplete with another one. Different results may also arise, if combined solvers share constraint symbols, depending on which solver comes first.
In the following we will adopt and extend the terminology and techniques of conditional term rewriting systems (CTRS) [DOS88, KK91]. A straightforward translation of results in the field of CTRS was not possible, because the CSR formalism gives rise to phenomena which do not appear in CTRS or make problems when treating confluence. These include the existence of global knowledge: CSR programs are more powerful than the classical conditional rewriting, because they use an additional context, the built-in constraint store. Information about this store must be available for application of computation steps. Other phenomena are: generalized, logical conditions for rule applicability (guards), multiple occurrences of variables on the left-hand side of a rule, local variables (variables that occur on the right-hand side of a rule only).
Confluence, as illustrated in Figure 3(A), guarantees that any computation starting from an arbitrary given initial state results in the same final state. We first define what it means that 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 CSR program is called *confluent*, if the following holds for all states S, S_1, S_2:

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

Example 3.1. Remember the following CSR program:

```
p ⇔ q.
p ⇔ false.
```

This program is obviously not confluent since `p` can either be replaced by `q` or `false` which differ. However the following program is confluent:

```
p ⇔ q.
p ⇔ false.
q ⇔ false.
```

Confluence is undecidable in general. Luckily, Newman's lemma [New42] for term rewriting systems is applicable to CSR as well: If a program is terminating, it suffices to consider local confluence to guarantee (global) confluence. We will show that local confluence is decidable for CSR (while termination, of course, is very likely to be undecidable).

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

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

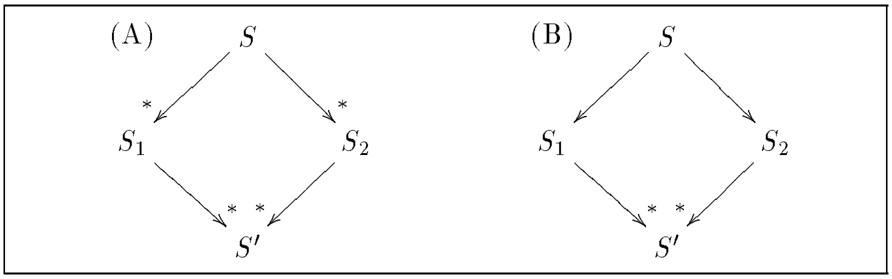

Figure 3. Confluence (A) and Local Confluence (B)

To analyze confluence of a given CSR program we have to check joinability of all pairs of states, which have a common ancestor state. There are infinitely many of those pairs, if there is at least one rule in the program. In the following we will present a decidable, necessary and sufficient condition for terminating CSR programs to be confluent. The idea of this criterion, as illustrated in Figure 4, is to test joinability of finitely many minimal pairs of states. These so-called critical pairs can be derived from rules overlapping heads. 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.

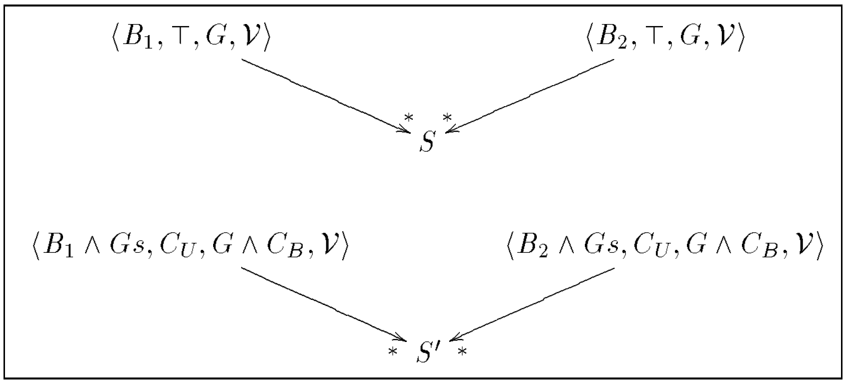

Figure 4. Joinability of Critical Pair (Top) and Extended States (Bottom)

Definition 3.4. If one or more head constraints $H_{i_1}, \ldots, H_{i_k}$ of a rule $(H_1, \ldots, H_n \Leftrightarrow G \mid B)$ can be equated with one or more head constraints $H'_{j_1}, \ldots, H'_{j_k}$ of a rule $(H'_1, \ldots, H'_m \Leftrightarrow G' \mid B')$[8], then we call the tuple

$$(\bar{G}, B \wedge H'_{j_{k+1}} \wedge \ldots \wedge H'_{j_m} =\downarrow= B' \wedge H_{i_{k+1}} \wedge \ldots \wedge H_{i_n}, \mathcal{V})$$

a *critical pair* of these rules. Here is $\bar{G} = G \wedge G' \wedge H_{i_1} \doteq H'_{j_1} \wedge \ldots \wedge H_{i_k} \doteq H'_{j_k}$, while $\{i_1, \ldots, i_n\}$ and $\{j_1, \ldots, j_m\}$ are permutations of $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$, respectively, and $1 \leq k \leq min(m, n)$. $\mathcal{V}$ is the sequence of variables in $H_1, \ldots, H_n$, $H'_1, \ldots, H'_m$.

Example 3.2. Consider the program for `maximum` of Example 2.4:

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

There are two trivial[9] and the following nontrivial critical pair[10]:

[8] It can be a fresh variant of the first rule.
[9] We call critical pairs of the form $(G\ ,\ B =\downarrow= B\ ,\ \mathcal{V})$ trivial.
[10] With variables from different rules already identified for readability.

$(\texttt{X}\leq\texttt{Y} \land \texttt{Y}\leq\texttt{X},\quad \texttt{Z}\doteq\texttt{Y} \;{=}{\downarrow}{=}\; \texttt{Z}\doteq\texttt{X},\quad \texttt{[X,Y,Z]})$

Example 3.3. Consider the following part of a CSR program defining interactions between the boolean operations `not`, `imp` and `or`.

$\texttt{not(X,Y), imp(X,Y)} \;\Leftrightarrow\; \texttt{true | X}\doteq\texttt{0, Y}\doteq\texttt{1.}$
$\texttt{not(X,Y), or(X,Z,Y)} \;\Leftrightarrow\; \texttt{true | X}\doteq\texttt{0, Y}\doteq\texttt{1, Z}\doteq\texttt{1.}$

These two rules have the nontrivial critical pair:

$\texttt{(true,}$
$\texttt{imp(X,Y)} \land \texttt{X}\doteq\texttt{0} \land \texttt{Y}\doteq\texttt{1} \land \texttt{Z}\doteq\texttt{1} \;{=}{\downarrow}{=}\; \texttt{or(X,Z,Y)} \land \texttt{X}\doteq\texttt{0} \land \texttt{Y}\doteq\texttt{1,}$
$\texttt{[X,Y,Z])}$

Definition 3.5. A critical pair $(G\ ,\ B_1 \;{=}{\downarrow}{=}\; B_2\ ,\ \mathcal{V})$ is called *joinable* if $\langle B_1, \top, G, \mathcal{V}\rangle$ and $\langle B_2, \top, G, \mathcal{V}\rangle$ are joinable.

Example 3.4. The critical pair in Example 3.2 is joinable, if the built-in constraint solver normalizes $\texttt{X}\leq\texttt{Y} \land \texttt{Y}\leq\texttt{X}$ into $\texttt{X}\doteq\texttt{Y}$. The critical pair in example 3.3 is also joinable, provided there are the following (or similar) rules in the CSR program:

$\texttt{imp(0,1)} \;\Leftrightarrow\; \texttt{true | true.}$
$\texttt{or(X,1,Z)} \;\Leftrightarrow\; \texttt{true | Z}\doteq\texttt{1.}$

With the notion of critical pairs we are in a position to give a sufficient and necessary condition for local confluence. The proof for the following theorem can be found in appendix B.

Theorem 3.1. A CSR program is locally confluent iff all its critical pairs are joinable.

Definition 3.6. A CSR program is called *terminating*, if there are no infinite computations.

The following corollary is a simple consequence of Theorem 3.1 and Newman's lemma [New42]:

Corollary 3.1. A terminating CSR program is confluent iff it is locally confluent.

The Corollary 3.1 gives a decidable characterization of confluent terminating CSR programs: Joinability of a given critical pair is decidable for a terminating CSR program and there are only finitely many critical pairs.
As in term rewriting systems, termination is crucial to go from local confluence to (global) confluence. It may be the case that the class of terminating

CSR programs is too restrictive to cover all programs that are useful in practice. But in the present we do not see a possibility to do without termination.

3.1. Confluence and Determinism

One may wonder why we need to bother with confluence instead of adopting the notion of determinism from ALPS [Mah87] to CSR. In a deterministic program, no two rules for the same predicate have overlapping guards. This means that in a computation, at most one rule can be chosen for a goal. Hence any possible order of rule applications results in the same final state. It may seem that any confluent program can be translated into an equivalent deterministic one. However, this is not the case, because the resulting deterministic programs may be operationally weaker than their confluent counterparts. The notion of deterministic programs is too strict for our purposes. The weakness of the notion of determinism applied to CSR has three reasons, of which the first two also hold for C^4L languages:
First, the constraint system must be closed under negation so that a C^4L program can be transformed into one without overlapping guards.

Example 3.5. Remember the (confluent) rules for `maximum`:

$$\texttt{maximum(X,Y,Z)} \Leftrightarrow \texttt{X}\leq\texttt{Y} \mid \texttt{Z}\doteq\texttt{Y}.$$
$$\texttt{maximum(X,Y,Z)} \Leftrightarrow \texttt{Y}\leq\texttt{X} \mid \texttt{Z}\doteq\texttt{X}.$$

This program cannot be transformed into an equivalent one without overlapping guards, if $\doteq$ and $\leq$ are the only built-in constraints.

Secondly, confluent programs can commit to a rule earlier than deterministic ones because their guards can be less rigid since they may overlap.

Example 3.6. A deterministic version of `maximum`:

$$\texttt{maximum(X,Y,Z)} \Leftrightarrow \texttt{X}<\texttt{Y} \mid \texttt{Z}\doteq\texttt{Y}.$$
$$\texttt{maximum(X,Y,Z)} \Leftrightarrow \texttt{Y}\leq\texttt{X} \mid \texttt{Z}\doteq\texttt{X}.$$

For the goal $\texttt{maximum(A,B,C)} \wedge \texttt{A}\leq\texttt{B}$ the answer is the goal itself, because no rule is applicable. In the confluent version (Example 3.5) the first rule commits and computes the answer $\texttt{A}\leq\texttt{B} \wedge \texttt{C}\doteq\texttt{B}$.

Third, in contrast to most C^4L languages including ALPS, CSR allow "multiple heads", i.e. conjunctions in the head of a rule. 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 to one of the rules (and thus making the program deterministic[11]) by adding constraints to the guards.

[11] We conservatively extend the notion of deterministic ALPS programs to CSR: At most one rule is applicable to any given goal.

Example 3.7. Consider the two rules of Example 3.3:

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

Given a goal `not(A,B)`$\wedge$`imp(A,B)`$\wedge$`or(A,C,B)`, both rules can be applied. To ensure that only the fist rule can be applied, we would have to add a condition to the guard of the first rule that `or(A,C,B)` does not exist in the current state. Such a condition cannot be expressed by a constraint since it is meta-logical as it can become dis-implied in the future.

4. Consistency and Confluence

We now show that confluence implies consistency of the logical meaning of a range-restricted program. For this we have to require the constraint theory to be ground complete[12]. Since our test for confluence is decidable, it thus can also be used to identify the parts of range-restricted terminating C^4L programs that have a consistent declarative semantics in our sense.
For the proof to go through, every rule has to satisfy a *range-restriction* condition: Every variable in the body appears also in the head. We believe that the result holds for general CSR programs, but to show this, it seems that a different proof technique has to be found.

Definition 4.1. A constraint theory CT is called *ground complete*, if for every ground atomic constraint c either $CT \models c$ or $CT \models \neg c$ holds.

Theorem 4.1. Let P be a range-restricted CSR program and CT a ground complete theory. If P is confluent, then $\mathcal{P} \cup CT$ is consistent.

The theorem follows directly from the following two lemmas. In order to formulate them, we first have to define the notion of computational equivalence:

Definition 4.2. Given a CSR program, we define the *computational equivalence* $\leftrightarrow^*$: $S_1 \leftrightarrow S_2$ iff $S_1 \mapsto S_2$ or $S_1 \mapsfrom S_2$. $S \leftrightarrow^* S'$ iff there is a sequence $S_1, \ldots, S_n$ such that S_1 is S, S_n is S' and $S_i \leftrightarrow S_{i+1}$ for all i.

We can easily see that for every computational equivalence $S \leftrightarrow^* S'$ there is a sequence $S_1, T_1, S_2, T_2, \ldots, T_{n-1}, S_n$ of the following form:

$$S \;{}^*\!\!\mapsfrom S_1 \mapsto^* T_1 \;{}^*\!\!\mapsfrom S_2 \mapsto^* \ldots \mapsto^* T_{n-1} \;{}^*\!\!\mapsfrom S_n \mapsto^* S'.$$

This sequence is more intuitively illustrated in Figure 5.

[12] Note that this restriction is very weak, since the property holds for almost all useful classes of constraint theories.

$$\begin{tikzcd}
& S_1 \arrow[dl, "*"'] \arrow[dr, "*"] & & S_2 \arrow[dl, "*"'] \arrow[dr, "*"] & & \cdots & & S_n \arrow[dl, "*"'] \arrow[dr, "*"] & \\
S & & T_1 & & T_2 & \cdots & T_{n-1} & & S'
\end{tikzcd}$$

Figure 5. Computational Equivalence of S and S'

Lemma 4.1. If P is confluent, then $\langle \top, \top, \text{true}, \mathcal{V}\rangle \leftrightarrow^* \langle \top, \top, \text{false}, \mathcal{V}\rangle$ does not hold.

Proof. Let be $S_T = \langle \top, \top, \text{true}, \mathcal{V}\rangle$ and $S_F = \langle \top, \top, \text{false}, \mathcal{V}\rangle$. We show by induction on the length of the computational equivalence n that there are no states $S_1, T_1, \ldots, T_{n-1}, S_n$ such that

$$S_T \; {}^{*}\!\mapsfrom S_1 \mapsto^* T_1 \; {}^{*}\!\mapsfrom \ldots \mapsto^* T_{n-1} \; {}^{*}\!\mapsfrom S_n \mapsto^* S_F$$

Base case: $S_T \; {}^{*}\!\mapsfrom S_1 \mapsto^* S_F$ cannot exist, because S_T and S_F are different final states and P is confluent.

Induction step: We assume that the induction hypothesis holds for n, i.e. $S_T \; {}^{*}\!\mapsfrom S_1 \mapsto^* T_1 \; {}^{*}\!\mapsfrom \ldots \mapsto^* T_{n-1} \; {}^{*}\!\mapsfrom S_n \mapsto^* S_F$ does not exist. We prove the assertion for $n+1$ by contradiction:

We assume that a sequence of the form

$S_T \; {}^{*}\!\mapsfrom S_1 \mapsto^* T_1 \; {}^{*}\!\mapsfrom \ldots \; {}^{*}\!\mapsfrom S_n \mapsto^* T_n \; {}^{*}\!\mapsfrom S_{n+1} \mapsto^* S_F$ exists. We will lead this assumption to a contradiction.

P is confluent, hence S_F and T_n are joinable. Since S_F is a final state, there is a computation of T_n that results in S_F (i.e. $T_n \mapsto^* S_F$), and hence $S_n \mapsto^* S_F$. Therefore there is a sequence of the form

$$S_T \; {}^{*}\!\mapsfrom S_1 \mapsto^* T_1 \; {}^{*}\!\mapsfrom \ldots \; {}^{*}\!\mapsfrom S_{n-1} \mapsto^* T_{n-1} \; {}^{*}\!\mapsfrom S_n \mapsto^* S_F,$$

which is a contradiction to the induction hypothesis. $\square$

Some notations and definitions are necessary before we go further. We use the notation "θ" for assignments, or valuations, to a set of variables. For an interpretation I and a variable valuation θ we denote the fact that the formula or set of formulas F is satisfied by I and θ as "$I, \theta \models F$". The fact that a closed formula is satisfied by an interpretation I is denoted as "$I \models F$".

An interpretation of $\mathcal{P} \cup CT$ is a structure that expands the Herbrand model of CT to include an interpretation of the set of the user-defined constraints appearing in the CSR program P. A model of a (set of) rule(s) is an interpretation modeling the rule (the set).

Lemma 4.2. Let P be a range-restricted CSR program and CT a ground complete theory. If $\langle \top, \top, \text{true}, \mathcal{V}\rangle \leftrightarrow^* \langle \top, \top, \text{false}, \mathcal{V}\rangle$ does not hold, then $\mathcal{P} \cup CT$ is consistent.

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

$$I_0 := \{\{C_1\theta, \ldots, C_n\theta\} \mid \langle (C_1 \wedge \ldots \wedge C_n)\theta, \top, \text{true}, \mathcal{V}\rangle \leftrightarrow^* \langle \top, \top, \text{true}, \mathcal{V}\rangle\}.$$

Let be $I := (\bigcup I_0)$.[13]
Because of the consistence and the ground completeness CT has a single Herbrand model

$$CM := \{c \mid CT \models c \text{ and } c \text{ is ground}\}$$

Let $\mathcal{I}$ be $I \cup CM$. We know that false $\notin I$, because $\langle \text{false}, \top, \text{true}, \mathcal{V}\rangle \leftrightarrow^* \langle \top, \top, \text{true}, \mathcal{V}\rangle$ does not hold. Therefore $\mathcal{I}$ is a Herbrand interpretation of $\mathcal{P} \cup CT$. We show $\mathcal{I} \models \mathcal{P}$.

Let $(H_1 \wedge \ldots \wedge H_n \Leftrightarrow G_1 \wedge \ldots \wedge G_j \mid B_1 \wedge \ldots \wedge B_k)$ be a CSR rule from P. We show $\mathcal{I} \models \forall\, ((G_1 \wedge \ldots \wedge G_j) \rightarrow (H_1 \wedge \ldots \wedge H_n \leftrightarrow \exists\, (B_1 \wedge \ldots \wedge B_k)))$. Since the rules are range-restricted, we have to show $\mathcal{I} \models \forall\, ((G_1 \wedge \ldots \wedge G_j) \rightarrow (H_1 \wedge \ldots \wedge H_n \leftrightarrow B_1 \wedge \ldots \wedge B_k))$
To show that $\mathcal{I} \models \forall\, (G_1 \wedge \ldots \wedge G_j) \rightarrow (H_1 \wedge \ldots \wedge H_n \leftrightarrow B_1 \wedge \ldots \wedge B_k)$ (which is equivalent to $\mathcal{I} \models \forall\, ((H_1 \wedge \ldots \wedge H_n \wedge G_1 \wedge \ldots \wedge G_j) \leftrightarrow (B_1 \wedge \ldots \wedge B_k \wedge G_1 \wedge \ldots \wedge G_j)))$, we have to show that $\mathcal{I}, \theta \models (H_1 \wedge \ldots \wedge H_n \wedge G_1 \wedge \ldots \wedge G_j \leftrightarrow B_1 \wedge \ldots \wedge B_k \wedge G_1 \wedge \ldots \wedge G_j)$ for any variable valuation θ.
For all formulas $(H_1 \wedge \ldots \wedge H_n \wedge G_1 \wedge \ldots \wedge G_j \leftrightarrow B_1 \wedge \ldots \wedge B_k \wedge G_1 \wedge \ldots \wedge G_j)$ and for any variable valuation θ the following equivalences hold:

$$\begin{aligned}
&\mathcal{I}, \theta \models H_1 \wedge \ldots \wedge H_n \wedge G_1 \wedge \ldots \wedge G_j \\
\text{iff } &\{H_1\theta, \ldots, H_n\theta, G_1\theta, \ldots, G_j\theta\} \subseteq \mathcal{I} \\
\text{iff } &\langle (H_1 \wedge \ldots \wedge H_n \wedge G_1 \wedge \ldots \wedge G_j)\theta, \top, \text{true}, \mathcal{V}\rangle \leftrightarrow^* \langle \top, \top, \text{true}, \mathcal{V}\rangle \\
\text{iff } &\langle (B_1 \wedge \ldots \wedge B_m \wedge G_1 \wedge \ldots \wedge G_j)\theta, \top, \text{true}, \mathcal{V}\rangle \leftrightarrow^* \langle \top, \top, \text{true}, \mathcal{V}\rangle \\
\text{iff } &\{B_1\theta, \ldots, B_m\theta, G_1\theta, \ldots, G_j\theta\} \subseteq \mathcal{I} \\
\text{iff } &\mathcal{I}, \theta \models B_1 \wedge \ldots \wedge B_m \wedge G_1 \wedge \ldots \wedge G_j.
\end{aligned}$$

Therefore $\mathcal{I}, \theta \models H_1 \wedge \ldots \wedge H_n \wedge G_1 \wedge \ldots \wedge G_j \leftrightarrow B_1 \wedge \ldots \wedge B_m \wedge G_1 \wedge \ldots \wedge G_j$ for any variable valuation θ and for all formulas in $\mathcal{P}$.
Then $\mathcal{I} \models \forall\, ((G_1 \wedge \ldots \wedge G_j) \rightarrow (H_1 \wedge \ldots \wedge H_n \leftrightarrow B_1 \wedge \ldots \wedge B_k))$ for all formulas from $\mathcal{P}$. □

[13] This operator denotes the union of all members of I_0.

5. Confluence and Declarative Semantics

The following theorem states that we can improve on completeness, if a CSR program is confluent and terminating.

Theorem 5.1. Let P be a terminating and confluent CSR program and G be a goal. Then the following are equivalent:

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

b) G has a computation with answer constraint C' such that $\mathcal{P}, CT \models \forall\ (C \leftrightarrow C')$.

c) Every computation of G has an answer constraint C' such that $\mathcal{P}, CT \models \forall\ (C \leftrightarrow C')$.

Proof. "a) $\Rightarrow$ b)" holds according to completeness of CSR computations, Theorem 2.2.
"b) $\Rightarrow$ c)" is implied directly by confluence and termination.
"c) $\Rightarrow$ a)" holds according to soundness of CSR computations, Theorem 2.1. □

The following theorem gives a condition for existence of finitely failed computations, provided the goals have the following property.

Definition 5.1. A goal is *data-sufficient*, if it has a computation with a final state containing an empty user-defined store.

This property guarantees that there is a computation of the goal with an answer constraint containing only built-in constraints. The property is exactly the same as the one used in [Mah87], but we use a more explicit definition[14].

Theorem 5.2. Let P be a range-restricted and confluent CSR program, CT a ground complete theory, and G a data-sufficient goal. If $\mathcal{P}, CT \models \neg\exists G$ then G has a finitely failed computation.
Proof. G has a computation with answer constraint C containing only built-in constraints, because G is data-sufficient.
By Theorem 2.1 the following does hold:

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

$\mathcal{P}, CT \models \neg\exists G$ implies $\mathcal{P}, CT \models \forall\ (\text{false} \leftrightarrow G)$. Therefore

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

[14] Personal communication with M. Maher, Email, January 1997.

Since P is confluent, $\mathcal{P}$ is consistent[15] (by Theorem 4.1). Furthermore, P does not define built-in constraints and C consists of built-in constraints only. Hence the following also holds:

$$CT \models \forall\ (C \leftrightarrow \text{false}).$$

Since C consists of built-in constraints only, C must be *false*. □

The following corollary is a soundness and completeness result for finite failure. It is a consequence of Theorems 5.1, 4.1 and 5.2.

Corollary 5.1. (Soundness and Completeness of Finite Failure) Let P be a range-restricted, terminating and confluent CSR program, CT a ground complete theory, and G a data-sufficient goal.
The following are equivalent:

a) $\mathcal{P}, CT \models \neg\exists G$

b) G has a finitely failed computation.

c) Every computation of G is finitely failed.

These results are similar to those for ALPS [Mah87], even though ALPS has a different declarative semantics (based on Clark's completion) and a different operational semantics (rules can commit more often).
As a conclusion of this section we present a comparison of the various completeness and soundness results for successful computations (SC) and finite failure (FF) for C^4L languages[16] as presented in [JM94] and CSR in Figure 6.

	C^4L	CSR	Determ. C^4L	Confl. CSR
Soundness (SC)	yes	yes	yes	yes
Completeness (SC)	no	yes	yes	yes
Soundness (FF)	yes	yes	yes	yes
Completeness (FF)	no	no	yes	yes

Figure 6. Soundness and Completeness Results for C^4L and CSR

[15] Thus, the proof also goes through for consistent programs.
[16] Note that the declarative semantics of these languages is different from CSR's (based on Clark's completion).

6. Conclusions and Future Work

We introduced the notion of confluence for Constraint Simplification Rules (CSR). Confluence guarantees that a CSR program consisting only of simplification rules will always compute the same result for a given set of user-defined constraints independent of which rules are applied.
Based on classical notions in term rewriting systems, we have given a characterization of confluence for terminating CSR programs through joinability of critical pairs, yielding a decidable, sufficient and necessary condition and syntactically based test for confluence. We have shown that confluence implies consistency of the logical meaning of CSR programs.
We also gave various soundness and completeness results for CSR programs. Our theorems are stronger than what holds for the related families of C^4L programming languages. Our approach complements recent work in program analysis as in [MO95, CFMW97], where a different, less rigid notion of confluence is defined: A committed-choice program is confluent, if different process schedulings give rise to the same set of possible outcomes. The idea of [MO95, CFMW97] is to introduce a non-standard semantics, which is confluent for all committed-choice programs.
We have developed a tool [Mar96] in $\mathrm{ECL}^i\mathrm{PS}^e$ (ECRC Constraint Logic Programming System [ACD$^+$94]) which tests confluence of CSR programs. Our tests show that most existing constraint solvers written in CSR are indeed confluent. A solver performing Gaussian elimination was not confluent. It can easily be made confluent by adding a condition to the guard (in this case, at the expense of efficiency). Current work [Abd97] integrates the two other kinds of CHR rules, the propagation and the simpagation rules, into our condition for confluence. The idea is to extend states by a component that keeps track of which propagation rules have already been applied and in this way avoids trivial nontermination.
As in term rewriting systems, termination is crucial to go from local confluence to (global) confluence. Thus investigations into termination are necessary.
We also want to investigate further the relationship of CSR to general-purpose C^4L languages. We plan to study completion methods to make a non-confluent CSR program confluent. Like in term rewriting systems, the idea is to turn critical pairs into rules.
Finally, we would like to thank the anonymous referees, who have pointed out some errors and omissions in preliminary versions 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, CP'97*, LNCS. Springer, November 1997.

ACD+94. A. Aggoun, D. Chan, P. Dufrense, E. Falvey, H. Grant, A. Herold, G. Macartney, M. Maier, D. Miller, B. Perez, E. van Rossum, J. Schimpf, P. Tsahageas, and D. de Villeneuve. *ECL^iPS^e 3.4 User Manual.* ECRC Munich Germany, July 1994.

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.

AKP94. Hassan Aït-Kaci and Andreas Podelski. Functions as passive constraints in LIFE. *ACM Transactions on Programming Languages and Systems*, 16(4):1279–1318, July 1994.

CD93. Philippe Codognet and Daniel Diaz. Boolean constraint solving using clp(FD). In Dale Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 525–539, Vancouver, Canada, 1993. The MIT Press.

CFMW97. M. Codish, M. Falaschi, K. Marriott, and W. Winsborough. A confluent semantic basis for the analysis of concurrent constraint logic programs. *Journal of Logic Programming*, 30(1):53–81, 1997.

Cla78. K. Clark. *Logic and Databases*, chapter Negation as Failure, pages 293–322. Plenum Press, New York, 1978.

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.

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

FHK+92. T. Frühwirth, A. Herold, V. Küchenhoff, T. Le Provost, P. Lim, E. Monfroy, and M. Wallace. Constraint logic programming: An informal introduction. In G. Comyn, N.E. Fuchs, and M.J. Ratcliffe, editors, *Logic Programming in Action*, LNCS 636, pages 3–35. Springer, 1992.

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

JL87. J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, 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. C. Kirchner and H. Kirchner. *Rewriting: Theory and Applications.* North-Holland, 1991.

Mah87. M. J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 858–876. The MIT Press, May 1987.

Mar96. M. Marte. Implementation eines Konfluenz-Tests für CSR-Programme. Advanced practical thesis, Institute of Computer Science, Ludwig-Maximilians-University Munich, 1996.

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

MO95. K. Marriott and M. Odersky. A confluent calculus for concurrent constraint programming with guarded choice. In *1st International Conference on Principles and Practice of Constraint Programming, CP'95*, pages 310–327. Springer, September 1995.

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

Pla93. D. 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. *ACM Computing Surveys*, 21(3):413–510, September 1989.

Smo91. G. Smolka. Residuation and guarded rules for constraint logic programming. In *Digital Equipment Paris Research Laboratory Research Report*, France, June 1991.

SRP91. V.A. Saraswat, M. Rinard, and P. Panangaden. The semantic foundations of concurrent constraint programming. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 333–352. ACM Press, January 1991.

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

Appendix

A. Proofs for Section 2.3

Lemma A.1. Let P be a CSR program, G be a goal. If C is a computable constraint of G, then

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

Proof. We prove the claim by structural induction over the computations:
Base case: No transition is applied to the initial state $\langle G, \top, \mathrm{true}, \mathcal{V}\rangle$, i.e. $C = G$, and the state is possibly normalized by $\mathcal{N}$. Then the following holds:

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

Induction step: We have the following computation

$$\langle G, \top, \mathrm{true}, \mathcal{V}\rangle \mapsto^* \langle Gs', C'_U, C'_B, \mathcal{V}\rangle \mapsto \langle Gs'', C''_U, C''_B, \mathcal{V}\rangle.$$

In order to prove that the last computation step preserves logical equivalence, we prove that each transition of the operational semantics preserves logical equivalence:
(1) **Solve**: Then Gs' is of the form $C \wedge Gs$, where C is a built-in constraint. The transition applied to the state $\langle C \wedge Gs, C'_U, C'_B, \mathcal{V}\rangle$ leads to the new state $\langle Gs'', C''_U, C''_B, \mathcal{V}\rangle = \mathcal{N}(\langle Gs, C'_U, C'_B \wedge C, \mathcal{V}\rangle)$.

Let $\bar{x}_1$ be the local variables of $\langle C \wedge Gs, C'_U, C'_B, \mathcal{V}\rangle$. By the induction hypothesis the following holds:

$$\mathcal{P}, CT \models \forall\ (\exists \bar{x}_1\ (C \wedge Gs \wedge C'_U \wedge C'_B) \leftrightarrow G).$$

The following holds by Lemma 2.1:

$$\mathcal{P}, CT \models \forall\ (\exists \bar{x}_1\ (C \wedge Gs \wedge C'_U \wedge C'_B) \leftrightarrow \exists \bar{x}_2 (Gs'' \wedge C''_U \wedge C''_B)),$$

where $\bar{x}_2$ are the local variables of $\langle Gs'', C''_U, C''_B, \mathcal{V}\rangle$.
Therefore

$$\mathcal{P}, CT \models \forall\ (\exists \bar{x}_2\ (Gs'' \wedge C''_U \wedge C''_B) \leftrightarrow G).$$

(2) **Introduce**: Then Gs' is of the form $C \wedge Gs$, where C is a user-defined constraint. **Introduce** applied to the state $\langle C \wedge Gs, C'_U, C'_B, \mathcal{V}\rangle$ leads to the new state $\langle Gs'', C''_U, C''_B, \mathcal{V}\rangle = \mathcal{N}(\langle Gs, C \wedge C'_U, C'_B, \mathcal{V}\rangle)$.
Let $\bar{x}_1$ be the local variables of $\langle C \wedge Gs, C'_U, C'_B, \mathcal{V}\rangle$. By the induction hypothesis the following holds:

$$\mathcal{P}, CT \models \forall\ (\exists \bar{x}_1\ (C \wedge Gs \wedge C'_U \wedge C'_B) \leftrightarrow G).$$

$\bar{x}_1$ are also the local variables of the state $\langle Gs, C \wedge C'_U, C'_B, \mathcal{V}\rangle$. The following equivalence holds:

$$\mathcal{P}, CT \models \forall\ (\exists \bar{x}_1\ (Gs \wedge C \wedge C'_U \wedge C'_B) \leftrightarrow G).$$

Since $\mathcal{N}$ does not change the state $\langle Gs, C \wedge C'_U, C'_B, \mathcal{V}\rangle$, x_2 are also the local variables of the state $\langle Gs'', C''_U, C''_B, \mathcal{V}\rangle$:

$$\mathcal{P}, CT \models \forall\ (\exists \bar{x}_2\ (Gs'' \wedge C''_U \wedge C''_B) \leftrightarrow G).$$

(3) **Simplify**: Then C'_U is of the form $H' \wedge C_U$, where $(H \Leftrightarrow C \mid B)$ is a fresh CSR rule from P and $CT \models \forall\ (C'_B \rightarrow \exists \bar{y}\ (H \doteq H' \wedge C))$.
The transition applied to the state $\langle Gs', H' \wedge C_U, C'_B, \mathcal{V}\rangle$ leads to the new state $\langle Gs'', C''_U, C''_B, \mathcal{V}\rangle = \mathcal{N}(\langle Gs' \wedge B, C_U, H \doteq H' \wedge C'_B, \mathcal{V}\rangle)$.
Let $\bar{x}_1$ be the local variables of $\langle Gs', H' \wedge C_U, C'_B, \mathcal{V}\rangle$. By the induction hypothesis the following holds:

$$\mathcal{P}, CT \models \forall\ (\exists x_1\ (Gs' \wedge H' \wedge C_U \wedge C'_B) \leftrightarrow G). \tag{4}$$

The entailment condition says that the context C'_B is equivalent to its conjunction with the instantiated guard.

$$CT \models \forall\ (C'_B \leftrightarrow C'_B \wedge \exists \bar{y}\ (H \doteq H' \wedge C))$$

$$CT \models \forall\, (C'_B \leftrightarrow \exists \bar{y}\, (C'_B \wedge H \doteq H' \wedge C))$$

By replacing C'_B by $\exists \bar{y}\, (C'_B \wedge H \doteq H' \wedge C)$ in equation (4) we obtain:

$$\mathcal{P}, CT \models \forall\, (\exists x_1\, (Gs' \wedge H' \wedge C_U \wedge \exists \bar{y}\, (C'_B \wedge H \doteq H' \wedge C)) \leftrightarrow G).$$

The variables $\bar{y}$ are the variables occurring only in H, then the following holds:

$$\mathcal{P}, CT \models \forall\, (\exists x_1\, \exists \bar{y}\, (Gs' \wedge H' \wedge C_U \wedge C'_B \wedge H \doteq H' \wedge C) \leftrightarrow G). \qquad (5)$$

According to the fact that $CT \models H \doteq H' \rightarrow (H \leftrightarrow H')$ we obtain:

$$\mathcal{P}, CT \models \forall\, (\exists x_1\, \exists \bar{y}\, (Gs' \wedge C_U \wedge C'_B \wedge H \doteq H' \wedge H \wedge C) \leftrightarrow G). \qquad (6)$$

From $\mathcal{P}, CT \models \forall\, ((C \rightarrow (H \leftrightarrow \exists \bar{y}_2\, B)))$ we deduce:

$$\mathcal{P}, CT \models \forall\, ((H \wedge C) \leftrightarrow \exists \bar{y}_2\, (B \wedge C))$$

By replacing $H \wedge C$ by $\exists \bar{y}_2\, (B \wedge C)$ in equation (6) we obtain:

$$\mathcal{P}, CT \models \forall\, (\exists x_1\, \exists \bar{y}\, (Gs' \wedge C_U \wedge C'_B \wedge H \doteq H' \wedge \exists \bar{y}_2\, (B \wedge C)) \leftrightarrow G). \qquad (7)$$

The variables $\bar{y}_2$ are the variables occurring only in B, then the following holds:

$$\mathcal{P}, CT \models \forall\, (\exists x_1\, \exists \bar{y}\, \exists \bar{y}_2\, (Gs' \wedge C_U \wedge C'_B \wedge H \doteq H' \wedge B \wedge C) \leftrightarrow G).$$

The entailment condition says that C is entailed by the context C'_B, then the following holds:

$$\mathcal{P}, CT \models \forall\, (\exists x_1\, \exists \bar{y}\, \exists \bar{y}_2\, (Gs' \wedge C_U \wedge C'_B \wedge H \doteq H' \wedge B) \leftrightarrow G).$$

The variables $\bar{y}_2$ (resp. $\bar{y}$) are the variables occurring only in B (resp. $H \doteq H'$), then $\bar{x}_2 = \bar{x}_1 \cup \bar{y}_2 \cup \bar{y}$ are the local variables of the state $\langle Gs' \wedge B, C_U, H \doteq H' \wedge C'_B, \mathcal{V} \rangle$:

$$\mathcal{P}, CT \models \forall\, (\exists \bar{x}_2\, (Gs' \wedge B \wedge C_U \wedge H \doteq H' \wedge C'_B) \leftrightarrow G).$$

Let $\bar{x}_3$ be the local variables of $\langle Gs'', C''_U, C''_B, \mathcal{V}\rangle$, then by Lemma 2.1:

$$\mathcal{P}, CT \models \forall\ (\exists \bar{x}_2\ (Gs' \wedge B \wedge C_U \wedge H \doteq H' \wedge C'_B) \leftrightarrow \exists \bar{x}_3\ (Gs'' \wedge C''_U \wedge C''_B)).$$

Therefore

$$\mathcal{P}, CT \models \forall\ (\exists \bar{x}_3\ (Gs'' \wedge C''_U \wedge C''_B) \leftrightarrow G)$$

holds. □

B. Proofs for Section 3

We first give the lemmas which are used in the proof of Theorem 3.1. Complete proofs for the lemmas are omitted for space reasons, they can be found in [Meu96].

The first lemma states when joinability is compatible with changing the global variable stores:

Lemma B.1. Let $\langle Gs_1, C_{U1}, C_{B1}, \mathcal{V}\rangle$ and $\langle Gs_2, C_{U2}, C_{B2}, \mathcal{V}\rangle$ be joinable. Then the following holds:

a) If $\mathcal{V}' \subseteq \mathcal{V}$, then $\langle Gs_1, C_{U1}, C_{B1}, \mathcal{V}'\rangle$ and $\langle Gs_2, C_{U2}, C_{B2}, \mathcal{V}'\rangle$ are joinable.

b) If $\mathcal{V}'$ contains only fresh variables, then $\langle Gs_1, C_{U1}, C_{B1}, \mathcal{V} \circ \mathcal{V}'\rangle$ and $\langle Gs_2, C_{U2}, C_{B2}, \mathcal{V} \circ \mathcal{V}'\rangle$ are joinable ($\circ$ denotes concatenation).

Proof sketch. a) If we reduce the number of global variables, there may be one effect on the computation steps: Variables that have been global before, are strictly local now. These variables will be eliminated by $\mathcal{N}$. Built-in constraints containing these variables will be changed to a representation without these variables. But the loss of information about these variables does not affect computation steps, because strictly local variables by definition do not appear anywhere else in the state. This is shown by induction over the number of computation steps.

b) This is shown by straightforward structural induction over computations.

The next lemma states that addition of constraints to the stores does not change joinability of the states. It is a consequence of monotonicity of logical consequence and of Lemma 2.2.

Lemma B.2. If $(C_1 \wedge C_2 \wedge C_3, Gs \wedge C_U \wedge C_B)$ is connected in $\mathcal{V}$ and

$$\langle Gs, C_U, C_B, \mathcal{V}\rangle \mapsto^* \langle Gs', C'_U, C'_B, \mathcal{V}\rangle,$$

then

$$\langle Gs \wedge C_1, C_U \wedge C_2, C_B \wedge C_3, \mathcal{V}\rangle \mapsto^* \\ \mathcal{N}(\langle Gs' \wedge C_1, C_U' \wedge C_2, C_B' \wedge C_3, \mathcal{V}\rangle).$$

The next lemma states that atoms can be moved from the goal store to the user-defined store without losing joinability.

Lemma B.3. If

$$\langle Gs_1 \wedge G_1, \top, C_{B1}, \mathcal{V}\rangle \text{ and } \langle Gs_2 \wedge G_2, \top, C_{B2}, \mathcal{V}\rangle$$

are joinable, and G_1 and G_2 are user-defined constraints, then

$$\langle Gs_1, G_1, C_{B1}, \mathcal{V}\rangle \text{ and } \langle Gs_2, G_2, C_{B2}, \mathcal{V}\rangle$$

are also joinable.

Proof. We divide $Gs_1 \wedge G_1$ into the built-in constraints G_{C1}, the user-defined constraints G_{STAT1}, which are not touched during the join, i.e. which remain in the goal store, and the user-defined constraints G_{MOVE1}, which are touched in the process of joining. Analogously, we divide $Gs_2 \wedge G_2$ into G_{C2}, G_{STAT2} and G_{MOVE2}.
In a first step, we show that, provided the requirement is met,

$$\langle G_{C1}, G_{MOVE1} \wedge G_{STAT1}, C_{B1}, \mathcal{V}\rangle \text{ and } \langle G_{C2}, G_{MOVE2} \wedge G_{STAT2}, C_{B2}, \mathcal{V}\rangle$$

are also joinable:
The only operation accessing the goal store with user-defined constraints is **Introduce**. Hence, all constraints G_{MOVE1} and G_{MOVE2}, respectively, are moved to the user-defined stores with **Introduce** steps during the process of joining. We can apply these **Introduce** steps in the beginning of the respective computation and append the remaining steps thereafter without changing the outcomes. Therefore $\langle G_{C1} \wedge G_{STAT1}, G_{MOVE1}, C_{B1}, \mathcal{V}\rangle$ and $\langle G_{C2} \wedge G_{STAT2}, G_{MOVE2}, C_{B2}, \mathcal{V}\rangle$ are also joinable, i.e. there are computation sequences for both states resulting in $\langle G \wedge G_{STAT1}, C_U, C_B, \mathcal{V}\rangle$ and $\langle G' \wedge G_{STAT2}, C_U', C_B', \mathcal{V}\rangle$, respectively, with these two states being variants. The same sequence of computation steps can be applied to the states $\langle G_{C1}, G_{MOVE1} \wedge G_{STAT1}, C_{B1}, \mathcal{V}\rangle$ and $\langle G_{C2}, G_{MOVE2} \wedge G_{STAT2}, C_{B2}, \mathcal{V}\rangle$, resulting in the states $\langle G, C_U \wedge G_{STAT1}, C_B, \mathcal{V}\rangle$ and $\langle G', C_U' \wedge G_{STAT2}, C_B', \mathcal{V}\rangle$ which are variants. This means that $\langle G_{C1}, G_{MOVE1} \wedge G_{STAT1}, C_{B1}, \mathcal{V}\rangle$ and $\langle G_{C2}, G_{MOVE2} \wedge G_{STAT2}, C_{B2}, \mathcal{V}\rangle$ are joinable.

In the second step, we show the assertion of the lemma:
$G_{MOVE1} \wedge G_{STAT1}$ represents the user-defined portion of $Gs_1 \wedge G_1$, and analogously $G_{MOVE2} \wedge G_{STAT2}$ represents the user-defined portion of $Gs_2 \wedge G_2$. Because G_1 and G_2 consist only of user-defined constraints, they are contained in $G_{MOVE1} \wedge G_{STAT1}$ and $G_{MOVE2} \wedge G_{STAT2}$, respectively. Hence, we can define G_{DIFF1} as the conjunction of user-defined constraints, so that $G_1 \wedge G_{DIFF1} = G_{MOVE1} \wedge G_{STAT1}$. (Remember that the conjunction is associative and commutative.) Analogously, we can define G_{DIFF2}.
We can deduce that $\langle G_{C1} \wedge G_{DIFF1}, G_1, C_{B1}, \mathcal{V}\rangle$ and $\langle G_{C1} \wedge G_{DIFF2}, G_2, C_{B2}, \mathcal{V}\rangle$ are joinable: If we a apply a series of **Introduce** steps to these states, we result in $\langle G_{C1}, G_{MOVE1} \wedge G_{STAT1}, C_{B1}, \mathcal{V}\rangle$ and $\langle G_{C2}, G_{MOVE2} \wedge G_{STAT2}, C_{B2}, \mathcal{V}\rangle$, respectively, which are joinable.
Because $G_{C1} \wedge G_{DIFF1} = Gs_1$, and $G_{C2} \wedge G_{DIFF2} = Gs_2$, we finally conclude, that

$$\langle Gs_1, G_1, C_{B1}, \mathcal{V}\rangle \text{ and } \langle Gs_2, G_2, C_{B2}, \mathcal{V}\rangle$$

are joinable. □

We are now in a position to prove the main theorem:
Proof of Theorem 3.1: "$\Longrightarrow$" direction: Let P be a locally confluent CSR program. We prove by contradiction that all critical pairs are joinable: Assume that $(G_1 \wedge G_2, B_1 \wedge H_2 =\downarrow= B_2 \wedge H_1, \mathcal{V})$ is a critical pair that is not joinable. We will construct a common ancestor state and then use the local confluence to contradict the assumption. With reordering the head constraints we can assume that this pair derives from the two rules

$$\begin{aligned} R_1 :\ H_1, H_3 &\Leftrightarrow G_1 \mid B_1 \\ R_2 :\ H_2, H_4 &\Leftrightarrow G_2 \mid B_2, \end{aligned}$$

where H_3 and H_4 can be equated[17]. Then

$$\langle B_1 \wedge H_2, \top, G_1 \wedge G_2, \mathcal{V}\rangle \text{ and}$$
$$\langle B_2 \wedge H_1, \top, G_1 \wedge G_2, \mathcal{V}\rangle$$

are not joinable and therefore

$$\langle B_1, H_2, G_1 \wedge G_2, \mathcal{V}\rangle \text{ and}$$
$$\langle B_2, H_1, G_1 \wedge G_2, \mathcal{V}\rangle$$

are not joinable.
Let $\langle \top, H_1' \wedge H_2' \wedge H_3', G_1' \wedge G_2', \mathcal{V}'\rangle$ be a fresh variant of $\langle \top, H_1 \wedge H_2 \wedge H_3, G_1 \wedge G_2, \mathcal{V}\rangle$.

[17] Remember that throughout the whole paper, the H_i denote conjunctions of atoms.

Now we can apply both R_1 and R_2 onto the state $\langle \top, H_1' \wedge H_2' \wedge H_3', G_1' \wedge G_2', \mathcal{V}' \rangle$ with the results

$$\mathcal{N}(\langle B_1'', H_2', G_1' \wedge G_2' \wedge H_1'' \doteq H_1' \wedge H_3'' \doteq H_3', \mathcal{V}' \rangle) \text{ and}$$
$$\mathcal{N}(\langle B_2'', H_1', G_1' \wedge G_2' \wedge H_2'' \doteq H_2' \wedge H_4'' \doteq H_3', \mathcal{V}' \rangle).$$

The fresh variants of the rules were $H_1'', H_3'' \Leftrightarrow G_1'' \mid B_1''$ and $H_2'', H_4'' \Leftrightarrow G_2'' \mid B_2''$. Because of the local confluence these states are joinable. We know that $\mathcal{N}$ propagates the equalities $H_1'' \doteq H_1' \wedge H_3'' \doteq H_3'$ and $H_2'' \doteq H_2' \wedge H_4'' \doteq H_3'$ into the respective goal store. This implies that we can substitute the variables in question, i.e. replace B_1'' and B_2'' by B_1' and B_2', respectively, without changing the outcome of $\mathcal{N}$.
If there are variables in B_1'' or B_2'' which do not occur in $H_1'' \wedge H_3''$ or $H_2'' \wedge H_4''$, respectively, i.e. variables whose values are not governed by the equalities $H_1'' \doteq H_1' \wedge H_3'' \doteq H_3'$ and $H_2'' \doteq H_2' \wedge H_4'' \doteq H_3'$, we do replace them by new variables, which does not influence the outcome of joinability.
Therefore

$$\mathcal{N}(\langle B_1', H_2', G_1' \wedge G_2' \wedge H_1'' \doteq H_1' \wedge H_3'' \doteq H_3', \mathcal{V}' \rangle) \text{ and}$$
$$\mathcal{N}(\langle B_2', H_1', G_1' \wedge G_2' \wedge H_2'' \doteq H_2' \wedge H_4'' \doteq H_3', \mathcal{V}' \rangle)$$

are joinable, too.
The following two states have the same normalized form as the upper states, and are as a simple consequence of Lemma 2.3 also joinable:

$$\langle B_1', H_2', G_1' \wedge G_2', \mathcal{V}' \rangle \text{ and}$$
$$\langle B_2', H_1', G_1' \wedge G_2', \mathcal{V}' \rangle,$$

This is a contradiction to the claim that the variant states

$$\langle B_1, H_2, G_1 \wedge G_2, \mathcal{V} \rangle \text{ and}$$
$$\langle B_2, H_1, G_1 \wedge G_2, \mathcal{V} \rangle$$

are not joinable.

"$\Longleftarrow$" direction: Let P be a CSR program where all critical pairs are joinable. We will show that P is locally confluent. Assume that we are in state S[18] where there are at least two different possibilities of computation:

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

We have to show that S_1 and S_2 are joinable. We investigate all pairs $S \mapsto S_1$ and $S \mapsto S_2$ and show that S_1 and S_2 are joinable. The joinability of critical pairs will play a central role in the case **Simplify** vs. **Simplify** only.

[18] Because of Lemma 2.3 we can assume that S is normalized.

Solve vs. Solve: Then S is of the form $\langle C_1 \wedge C_2 \wedge Gs, C_U, C_B, \mathcal{V}\rangle$. We can move two different built-in constraints from the goal store to the built-in store:

$$\begin{aligned}\langle C_1 \wedge C_2 \wedge Gs, C_U, C_B, \mathcal{V}\rangle &\mapsto \mathcal{N}(\langle C_2 \wedge Gs, C_U, C_B \wedge C_1, \mathcal{V}\rangle) \\ &= \langle C_2' \wedge Gs_1, C_{U1}, C_{B1}, \mathcal{V}\rangle \text{ and} \\ \langle C_1 \wedge C_2 \wedge Gs, C_U, C_B, \mathcal{V}\rangle &\mapsto \mathcal{N}(\langle C_1 \wedge Gs, C_U, C_B \wedge C_2, \mathcal{V}\rangle) \\ &= \langle C_1' \wedge Gs_2, C_{U2}, C_{B2}, \mathcal{V}\rangle.\end{aligned}$$

It is easy to see that we can apply the other **Solve** step onto each resulting state. It is obvious that the resulting states will be identical:

$$\mathcal{N}(\langle Gs_1, C_{U1}, C_{B1} \wedge C_2', \mathcal{V}\rangle) = \mathcal{N}(\langle Gs_2, C_{U2}, C_{B2} \wedge C_1', \mathcal{V}\rangle).$$

Solve vs. Simplify: S is of the form $\langle C \wedge Gs, H' \wedge C_U, C_B, \mathcal{V}\rangle$, 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 G of the rule is implied by C_B.
Application of **Simplify** results in $S_{SIMP} = \langle B' \wedge C' \wedge Gs', C_U', C_B', \mathcal{V}\rangle = \mathcal{N}(\langle B \wedge C \wedge Gs, C_U, C_B \wedge H \doteq H', \mathcal{V}\rangle)$, whereas application of **Solve** leads to $S_{SOLVE} = \langle Gs'', H'' \wedge C_U'', C_B'', \mathcal{V}\rangle = \mathcal{N}(\langle Gs, H' \wedge C_U, C \wedge C_B, \mathcal{V}\rangle)$.
Of course, **Solve** is applicable on S_{SIMP}, resulting in $S_{END} = \mathcal{N}(\langle B' \wedge Gs', C_U', C' \wedge C_B', \mathcal{V}\rangle)$.
Application of **Simplify** on S_{SOLVE} is possible, because $CT \models C_B'' \leftrightarrow \forall\ \exists \bar{x}(C \wedge C_B)$ ($\bar{x}$ strictly local in $\langle Gs'', H'' \wedge C_U'', C \wedge C_B, \mathcal{V}\rangle$) and $CT \models \forall(C_B \rightarrow \exists \bar{y}(G \wedge H \doteq H'))$ ($\bar{y}$ are the variables in $(H \Leftrightarrow G \mid B)$), therefore $CT \models \forall(C_B'' \rightarrow \exists \bar{y}(G \wedge H \doteq H''))$ holds. This results in $S_{END}' = \mathcal{N}(\langle B \wedge Gs'', C_U'', C_B'' \wedge H \doteq H'', \mathcal{V}\rangle)$, which is in fact identical to S_{END}:
$CT \models (\exists \bar{z}_1\ C' \wedge C_B') \leftrightarrow (\exists \bar{z}_2\ C_B'' \wedge H \doteq H'')$, where $\bar{z}_1$ and $\bar{z}_2$ are the strictly local variables in $\langle B' \wedge Gs', C_U', C' \wedge C_B', \mathcal{V}\rangle$ and $\langle B \wedge Gs'', C_U'', C_B'' \wedge H \doteq H'', \mathcal{V}\rangle$, respectively, must hold, because of the following two equivalences which are guaranteed by the projection property of $\mathcal{N}$:

$$\begin{aligned}CT &\models \forall(C_B' \leftrightarrow \exists \bar{x}_1\ C_B \wedge H \doteq H') \\ CT &\models \forall(C_B'' \leftrightarrow \exists \bar{x}\ C_B \wedge C).\end{aligned}$$

(Analysis of the strictly local variables of the respective states leads to $CT \models (\exists \bar{z}_1\ C' \wedge C_B') \leftrightarrow (\exists \bar{z}_2\ C_B'' \wedge H \doteq H'')$.)
Because of the uniqueness of $\mathcal{N}$, the built-in states of S_{END} and S_{END}' are identical. According to equality propagation of $\mathcal{N}$, S_{END} and S_{END}' have identical goal and user-defined stores.
Introduce vs. Introduce: We know that S must be of the form $\langle C_1 \wedge C_2 \wedge Gs, C_U, C_B, \mathcal{V}\rangle$ where C_1 and C_2 are user-defined constraints.

Obviously the results of two computations are identical, since conjuncts can be permuted.

Introduce vs. Simplify: S is of the form $\langle C \wedge Gs, H \wedge C_U, C_B, \mathcal{V}\rangle$, where C is a user-defined constraint, and H a conjunction of constraints matching with the head of a rule $(H_1 \Leftrightarrow G|B)$. The two successor states are

$$\begin{aligned}
\textbf{Introduce}: S_1 &= \mathcal{N}(\langle Gs, C \wedge H \wedge C_U, C_B, \mathcal{V}\rangle) \\
&= \langle Gs, C \wedge H \wedge C_U, C_B, \mathcal{V}\rangle \text{ and} \\
\textbf{Simplify}: S_2 &= \mathcal{N}(\langle C \wedge Gs \wedge B, C_U, C_B \wedge H = H_1, \mathcal{V}\rangle) \\
&= \langle C' \wedge Gs' \wedge B', C'_U, C'_B, \mathcal{V}\rangle.
\end{aligned}$$

The second of the four equations holds, because we assumed that S was normalized. We can apply the other computation step onto S_1 and S_2 resulting in:

$$\begin{aligned}
S'_1 &= \mathcal{N}(\langle Gs \wedge B, C \wedge C_U, C_B \wedge H = H_1, \mathcal{V}\rangle) \\
&= \langle Gs' \wedge B', C' \wedge C'_U, C'_B, \mathcal{V}\rangle \text{ and} \\
S'_2 &= \mathcal{N}(\langle Gs' \wedge B', C' \wedge C'_U, C'_B, \mathcal{V}\rangle) \\
&= \langle Gs' \wedge B', C' \wedge C'_U, C'_B, \mathcal{V}\rangle.
\end{aligned}$$

The second equation follows from the fourth equation in the equations above. The fourth equation holds, because S_2 was normalized.
This means that $S'_1 = S'_2$, i.e. S_1 and S_2 are joinable.

Introduce vs. Solve: This situation is analogous to the case **Introduce** vs. **Simplify**.

Simplify vs. Simplify: Let be

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

the rules[19] being applied to the state S. We have to show that application of R or R' onto the state S results in joinable states. We know that the built-in store C_B of S is satisfiable, otherwise no rule could be applied. We can distinguish two different subcases:
Disjoint Peak: No constraint H_i of the head of the rule R can be equated with a constraint H_j of the head of the other rule R'. Obviously the two rules can be applied in any order, since they replace different conjuncts.
Critical Peak: In order to show joinability of S_1 and S_2, we will use the

[19] R and R' can be fresh variants of the same rule.

assumption that all critical pairs are joinable, find a critical pair that corresponds to S_1 and S_2 and modify the two involved states step by step, whilst keeping joinability, until we know in the end that S_1 and S_2 are joinable.
Without loss of generality we can change the order of the head atoms in R and R'. We can assume that the variables in the CSR program are disjoint from the variables in the actual state S. Now we are in a situation where the first atoms of the rules can be equated (i.e. $CT \models \exists (H_1 \dot{=} H'_1 \wedge \ldots \wedge H_i \dot{=} H'_i)$ where $1 \leq i \leq n$ and $i \leq n'$).
Let $S = \langle Gs, G_1 \wedge \ldots \wedge G_m, C_B, \mathcal{V} \rangle$ be the actual state, on which the rules R and R' are applicable. In order to be applied, the conditions of **Simplify** must be fulfilled, i.e. $CT \models \forall\ (C_B \rightarrow \exists \bar{x} C_1)$ and $CT \models \forall\ (C_B \rightarrow \exists \bar{y} C_2)$, where C_1 and C_2 are the conjunctions of the respective guard with the equality constraints derived from the matching, i.e. the following conjunction of constraints:

$$\begin{aligned} C_1 &= G \wedge G_1 \dot{=} H_1 \wedge \ldots \wedge G_i \dot{=} H_i \wedge G_{i+1} \dot{=} H_{i+1} \wedge \ldots \wedge G_n \dot{=} H_n, \\ C_2 &= G' \wedge G_1 \dot{=} H'_1 \wedge \ldots \wedge G_i \dot{=} H'_i \wedge G_{n+1} \dot{=} H'_{i+1} \wedge \ldots \wedge G_{n+n'-i} \dot{=} H'_n. \end{aligned}$$

We use abbreviations to represent the atoms in question:

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

$\bar{H}$ and $\bar{H}'$ are the heads of R and R', $\bar{G}$ (resp. $\bar{G}'$) are the matching constraints of the user-defined store in S with H (resp. H'), $\bar{H}_\cap$ and $\bar{H}'_\cap$ are the common parts of $\bar{H}$ and $\bar{H}'$ (i.e. the overlapping constraints of the rule heads), and $\bar{G}_R$ and $\bar{G}'_R$ represent the contents of the user-defined store after removing the matching constraints $\bar{G}$ and $\bar{G}'$, respectively.
The application of R and R', respectively, on the actual state will result in the following two states:

$$\begin{aligned} S_1 &\equiv \mathcal{N}(\langle Gs \wedge B, \bar{G}_R, C_B \wedge \bar{G} \dot{=} \bar{H}, \mathcal{V} \rangle) \\ S_2 &\equiv \mathcal{N}(\langle Gs \wedge B', \bar{G}'_R, C_B \wedge \bar{G}' \dot{=} \bar{H}', \mathcal{V} \rangle) \end{aligned}$$

We will show in the following that S_1 and S_2 are joinable.
We can see that the rules R and R' have the critical pair

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

We will use C_{JOIN} as an abbreviation for the joined guard, $G \wedge G' \wedge \bar{H}_\cap \doteq \bar{H}'_\cap$. We know that this critical pair is joinable. So there is a computation resulting in two final states S'_1 and S'_2 where S'_1 and S'_2 are variants:

$$\begin{aligned} &\mathcal{N}(\langle B \wedge H'_{i+1} \wedge \ldots \wedge H'_{n'}, \top, C_{JOIN}, \mathcal{V}' \rangle) \mapsto^* S'_1, \\ &\mathcal{N}(\langle B' \wedge H_{i+1} \wedge \ldots \wedge H_n, \top, C_{JOIN}, \mathcal{V}' \rangle) \mapsto^* S'_2. \end{aligned}$$

We can apply Lemma B.1 b) here and add $\mathcal{V}$ to the global variables stores, because by our assumption $\mathcal{V}$ shares no variables with the two states:

$$\begin{aligned} &\mathcal{N}(\langle B \wedge H'_{i+1} \wedge \ldots \wedge H'_{n'}, \top, C_{JOIN}, \mathcal{V}' \circ \mathcal{V} \rangle) \text{ and} \\ &\mathcal{N}(\langle B' \wedge H_{i+1} \wedge \ldots \wedge H_n, \top, C_{JOIN}, \mathcal{V}' \circ \mathcal{V} \rangle) \end{aligned}$$

are joinable.

By Lemma B.2 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 $G_{n+n'-i+1} \wedge \ldots \wedge G_m$ 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, by assumption, $G_{n+n'-i+1} \wedge \ldots \wedge G_m$ share no variables with the goal, user-defined and built-in stores of the previous states.

$$\begin{aligned} &\mathcal{N}(\langle Gs \wedge B \wedge H'G, \top, C_{JOIN} \wedge C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}' \circ \mathcal{V} \rangle) \text{ and} \\ &\mathcal{N}(\langle Gs \wedge B' \wedge HG, \top, C_{JOIN} \wedge C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}' \circ \mathcal{V} \rangle) \end{aligned}$$

are joinable. Here $H'G$ stands for $H'_{i+1} \wedge \ldots \wedge H'_{n'} \wedge G_{n+n'-i+1} \wedge \ldots \wedge G_m$ and HG stands for $H_{i+1} \wedge \ldots \wedge H_n \wedge G_{n+n'-i+1} \wedge \ldots \wedge G_m$.

Now we can remove the global variables $\mathcal{V}'$ from the variable stores by applying Lemma B.1 a) and keep joinability of

$$\mathcal{N}(\langle Gs \wedge B \wedge H'G, \top, C_{JOIN} \wedge C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V} \rangle) \text{ and} \tag{8}$$

$$\mathcal{N}(\langle Gs \wedge B' \wedge HG, \top, C_{JOIN} \wedge C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V} \rangle). \tag{9}$$

Because both R and R' are applicable to the state S, we know that:

$$CT \models \forall \; (C_B \rightarrow \exists \bar{x} \exists \bar{y} (G \wedge G' \wedge \bar{H}_\cap \doteq \bar{H}'_\cap))$$

implying that

$$\begin{aligned} CT \models \forall \quad & (C_B \wedge \exists \bar{x} \exists \bar{y} (\bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}') \\ & \leftrightarrow C_B \wedge \exists \bar{x} \exists \bar{y} (G \wedge G' \wedge \bar{H}_\cap \doteq \bar{H}'_\cap \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}')) \end{aligned}$$

The uniqueness property of $\mathcal{N}$ implies that

$$\begin{aligned} &\mathcal{N}(\langle Gs \wedge B \wedge H'G, \top, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V} \rangle) \text{ and} \\ &\mathcal{N}(\langle Gs \wedge B' \wedge HG, \top, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V} \rangle) \end{aligned}$$

are identical to the states (8) and (9) and therefore joinable.
We propagate the equalities $\bar{G}' \doteq \bar{H}'$ in the built-in stores to the rest of the states. This results in states with the same normalized form. Therefore 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 states again:

$$\mathcal{N}(\langle Gs \wedge B \wedge \bar{G}_R, \top, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}\rangle) \text{ and}$$
$$\mathcal{N}(\langle Gs \wedge B' \wedge \bar{G}'_R, \top, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}\rangle).$$

Let $\bar{x}$ be the variables of $\bar{H}'$. By our assumption the variables $\bar{x}$ in $\langle Gs \wedge B \wedge \bar{G}_R, \top, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}\rangle$ are strictly local, and the following holds (because the constraint $\bar{G}' \doteq \bar{H}'$ means that $\bar{G}'$ is an instance of $\bar{H}'$):

$$CT \models \forall\ (\exists \bar{x}(C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}') \leftrightarrow (C_B \wedge \bar{G} \doteq \bar{H})).$$

The uniqueness property of $\mathcal{N}$ and a likewise reasoning for the variables of $\bar{H}$ then lead to:

$$\begin{array}{ll} & \mathcal{N}(\langle Gs \wedge B \wedge \bar{G}_R, \top, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}\rangle) \\ = & \mathcal{N}(\langle Gs \wedge B \wedge \bar{G}_R, \top, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{V}\rangle) \\ \text{and} & \\ & \mathcal{N}(\langle Gs \wedge B' \wedge \bar{G}'_R, \top, C_B \wedge \bar{G} \doteq \bar{H} \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}\rangle) \\ = & \mathcal{N}(\langle Gs \wedge B' \wedge \bar{G}'_R, \top, C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}\rangle). \end{array}$$

This implies that the states

$$\mathcal{N}(\langle Gs \wedge B \wedge \bar{G}_R, \top, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{V}\rangle) \text{ and}$$
$$\mathcal{N}(\langle Gs \wedge B' \wedge \bar{G}'_R, \top, C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}\rangle)$$

are joinable. By applying Lemma B.3 we finally know that

$$\mathcal{N}(\langle Gs \wedge B, \bar{G}_R, C_B \wedge \bar{G} \doteq \bar{H}, \mathcal{V}\rangle) \text{ and}$$
$$\mathcal{N}(\langle Gs \wedge B', \bar{G}'_R, C_B \wedge \bar{G}' \doteq \bar{H}', \mathcal{V}\rangle)$$

are joinable. $\square$